Radhia Cousot
Matthieu Martel (Eds.)

ARCoSS

LNCS 6337

Static Analysis

17th International Symposium, SAS 2010
Perpignan, France, September 2010
Proceedings

Springer

Lecture Notes in Computer Science 6337

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

Editorial Board

David Hutchison, UK
Takeo Kanade, USA
Josef Kittler, UK
Jon M. Kleinberg, USA
Alfred Kobsa, USA
Friedemann Mattern, Switzerland
John C. Mitchell, USA
Moni Naor, Israel
Oscar Nierstrasz, Switzerland
C. Pandu Rangan, India
Bernhard Steffen, Germany
Madhu Sudan, USA
Demetri Terzopoulos, USA
Doug Tygar, USA
Gerhard Weikum, Germany

Advanced Research in Computing and Software Science

Subline of Lectures Notes in Computer Science

Subline Series Editors

Giorgio Ausiello, *University of Rome 'La Sapienza', Italy*
Vladimiro Sassone, *University of Southampton, UK*

Subline Advisory Board

Susanne Albers, *University of Freiburg, Germany*
Benjamin C. Pierce, *University of Pennsylvania, USA*
Bernhard Steffen, *University of Dortmund, Germany*
Madhu Sudan, *Microsoft Research, Cambridge, MA, USA*
Deng Xiaotie, *City University of Hong Kong*
Jeannette M. Wing, *Carnegie Mellon University, Pittsburgh, PA, USA*

Radhia Cousot   Matthieu Martel (Eds.)

Static Analysis

17th International Symposium, SAS 2010
Perpignan, France, September 14-16, 2010
Proceedings

Springer

Volume Editors

Radhia Cousot
École normale supérieure
Départment d'Informatique
45, rue d'Ulm
75230 Paris cedex, France
E-mail: radhia.cousot@ens.fr

Matthieu Martel
Université de Perpignan Via Domitia
ELIAUS-DALI Laboratory
52, avenue Paul Alduy
66860 Perpignan cedex, France
E-mail: matthieu.martel@univ-perp.fr

Library of Congress Control Number: 2010934118

CR Subject Classification (1998): D.2, F.3, D.3, D.2.4, F.4.1, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-15768-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-15768-4 Springer Berlin Heidelberg New York

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

springer.com

© Springer-Verlag Berlin Heidelberg 2010
Printed in Germany

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

Preface

Static analysis is a research area aimed at developing principles and tools for verification, certification, semantics-based manipulation, and high-performance implementation of programming languages and systems. The series of Static Analysis Symposia has served as the primary venue for presentation and discussion of theoretical, practical, and applicational advances in the area.

This year's symposium, The 17th International Static Analysis Symposium (SAS 2010), was held on September 14–16, 2010 in Perpignan, France, with 3 affiliated workshops: NSAD 2010 (The Second Workshop on Numerical and Symbolic Abstract Domains), SASB 2010 (The First Workshop on Static Analysis and Systems Biology) on September 13, 2010, and TAPAS 2010 (Tools for Automatic Program Analysis) on September 17, 2010.

The programme of SAS 2010 included a special session dedicated to the memory of the outstanding computer scientists *Robin Milner* and *Amir Pnueli.* This session consisted of 5 invited talks by E. Allen Emerson (The University of Texas at Austin, USA), Benjamin Goldberg (New York University, USA), James Leifer (INRIA Paris–Rocquencourt, France), Joachim Parrow (Uppsala University, Sweden), and Glynn Winskel (University of Cambridge, UK).

There were 58 submissions. Each submission was reviewed by at least three programme committee members. The committee decided to accept 22 papers.

In addition to the special session and the 22 contributed papers, the programme included 4 invited talks by Manuel Fähndrich (Microsoft Research, USA), David Lesens (EADS Space Transportation, France), Andreas Podelski (Freiburg University, Germany), and Mooly Sagiv (Tel-Aviv University, Israel and Stanford University, USA).

We would like to thank all the external referees for their participation in the reviewing process. Special thanks to Albertine Martel for the design of the nice poster and websites of SAS 2010 and its affiliated workshops.

We are grateful to our generous sponsors (CNRS, École Normale Supérieure, INRIA, Microsoft Research, and University of Perpignan), to all the members of the Organizing Committee in Perpignan, and to the Easy Chair team for the use of their very handy system.

June 2010

Radhia Cousot
Matthieu Martel

Conference Organization

Programme Chairs

Radhia Cousot	École Normale Supérieure and CNRS, France
Matthieu Martel	Université de Perpignan Via Domitia, France

Programme Committee

Elvira Albert	Complutense University of Madrid, Spain
Mariá Alpuente	Universidad Politecnica de Valencia, Spain
Olivier Bouissou	Commissariat à l'Energie Atomique, France
Byron Cook	Microsoft Research, Cambridge, UK
Susanne Graf	Verimag/CNRS, Grenoble, France
Joxan Jaffar	National University of Singapore, Singapore
Neil Jones	University of Copenhagen, Denmark
Francesca Levi	University of Pisa, Italy
Francesco Logozzo	Microsoft Research, Redmond, USA
Damien Massé	Université de Bretagne Occidentale, France
Isabella Mastroeni	Università degli Studi di Verona, Italy
Laurent Mauborgne	École Normale Supérieure, France
Matthew Might	University of Utah, USA
George Necula	University of California, Berkeley, USA
Francesco Ranzato	Università di Padova, Italy
Andrey Rybalchenko	Max Planck Institute, Saarbrucken, Germany
Mary Lou Soffa	University of Virginia, USA
Zhendong Su	University of California, Davis, USA
Greta Yorsh	IBM T.J. Watson Research Center, USA

Local Arrangement Chair

Matthieu Martel

External Reviewers

Aaron Bradley, Adriano Peron, Alessandra Di Pierro, Alexandre Chapoutot, Alexandre Donze, Alexey Gotsman, Alicia Villanueva, Andrea Masini, Andrea Turrini, Andrew Santosa, Anindya Banerjee, Antoine Miné, Ashutosh Gupta, Assalí Adjí, Axel Simon, Bertrand Jeannet, Chiara Bodei, Chris Hankin, Christophe Joubert, Christos Stergiou, Damiano Zanardini, Daniel Grund, David Pichardie, David Sanan, Dino Distefano, Durica Nikolic, Earl Barr, Enea Zaffanella, Enric Carbonell, Eran Yahav, Eric Koskinen, Fausto Spoto, Florent Bouchez, Francesca Scozzari, Francesco Tapparo, Francois Pottier, Georges Gonthier, Guido Scatena, Hongseok Yang, Jacob Burnim, Jacob Howe, Jan Olaf Blech, Jan Reineke, Jérôme Feret, Jorge Navas, Josh Berdine, Juan Chen, Julien Bertrane, Khalil Ghorbal, Laura Ricci, Leonardo de Moura, Liang Xu, Mario Mendez-Lojo, Marisa Llorens, Mark Gabel, Mark Marron, Massimo Merro, Matko Botincan, Matthew Parkinson, Michaël Monerau, Moreno Falaschi, Musard Balliu, Nicholas Kidd, Patricia M. Hill, Pierre Ganty, Raluca Sauciuc, Ranjit Jhala, Razvan Voicu, Ricardo Peá, Roberto Giacobazzi, Roland Yap, Salvador Lucas, Samir Genaim, Santiago Escobar, Sarah Zennou, Silvia Crafa, Songtao Xia, Sriram Sankaranarayanan, Stephen Magill, Stéphane Devismes, Techio Terauchi, Thomas Wies, Tim Harris, VIktor Vafeiadis, Virgile Prevosto, Wei-Ngan Chin, Xavier Rival, and Yishai A. Feldman.

Table of Contents

Time of Time (Invited Talk) 1
E. Allen Emerson

Static Verification for Code Contracts (Invited Talk) 2
Manuel Fähndrich

Translation Validation of Loop Optimizations and Software Pipelining in the TVOC Framework: In Memory of Amir Pnueli (Invited Talk) 6
Benjamin Goldberg

Size-Change Termination and Transition Invariants (Invited Talk)...... 22
Matthias Heizmann, Neil D. Jones, and Andreas Podelski

Using Static Analysis in Space: Why Doing so? (Invited Talk) 51
David Lesens

Statically Inferring Complex Heap, Array, and Numeric Invariants (Invited Talk) 71
Bill McCloskey, Thomas Reps, and Mooly Sagiv

From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis 100
Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Diana Vanessa Ramírez Deantes

Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs 117
Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord

Deriving Numerical Abstract Domains via Principal Component Analysis 134
Gianluca Amato, Maurizio Parton, and Francesca Scozzari

Concurrent Separation Logic for Pipelined Parallelization 151
Christian J. Bell, Andrew W. Appel, and David Walker

Automatic Abstraction for Intervals Using Boolean Formulae 167
Jörg Brauer and Andy King

Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables 184
Alexandre Chapoutot

A Shape Analysis for Non-linear Data Structures 201
Renato Cherini, Lucas Rearte, and Javier Blanco

Modelling Metamorphism by Abstract Interpretation 218
Mila Dalla Preda, Roberto Giacobazzi, Saumya Debray, Kevin Coogan, and Gregg M. Townsend

Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis 236
Isil Dillig, Thomas Dillig, and Alex Aiken

Compositional Bitvector Analysis for Concurrent Programs with Nested Locks 253
Azadeh Farzan and Zachary Kincaid

Computing Relaxed Abstract Semantics w.r.t. Quadratic Zones Precisely 271
Thomas Martin Gawlitza and Helmut Seidl

BOXES: A Symbolic Abstract Domain of Boxes 287
Arie Gurfinkel and Sagar Chaki

Alternation for Termination 304
William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani

Interprocedural Analysis with Lazy Propagation 320
Simon Holm Jensen, Anders Møller, and Peter Thiemann

Verifying a Local Generic Solver in Coq 340
Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl

Thread-Modular Counterexample-Guided Abstraction Refinement 356
Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko

Generating Invariants for Non-linear Hybrid Systems by Linear Algebraic Methods 373
Nadir Matringe, Arnaldo Vieira Moura, and Rachid Rebiha

Linear-Invariant Generation for Probabilistic Programs: Automated Support for Proof-Based Methods 390
Joost-Pieter Katoen, Annabelle K. McIver, Larissa A. Meinicke, and Carroll C. Morgan

Abstract Interpreters for Free 407
Matthew Might

Points-to Analysis as a System of Linear Equations 422
Rupesh Nasre and Ramaswamy Govindarajan

Strictness Meets Data Flow 439
Tom Schrijvers and Alan Mycroft

Automatic Verification of Determinism for Structured Parallel Programs 455
Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar

Author Index 473

Time of Time

E. Allen Emerson[1,2]

[1] Department of Computer Science
[2] Computer Engineering Research Center
The University of Texas at Austin,
Austin TX 78712, USA

Abstract. In his landmark 1977 paper "The Temporal Logic of Programs", Amir Pnueli gave a fundamental recognition that the ideally nonterminating behavior of ongoing concurrent programs, such as operating systems and protocols, was a vital aspect of program reasoning. As classical approaches to program correctness were based on initial-state/final-state semantics for terminating programs, these approaches were inapplicable to programs where infinite behavior was the norm. To address this shortcoming, Pnueli suggested the use of *temporal logic*, a formalism for reasoning about change over time originally studied by philosophers, to meaningfully describe and reason about the infinite behavior of programs. This suggestion turned out to be remarkably fruitful. It struck a resonant chord within the formal verification community, and it has had an enormous impact on the development of the area. It matured into an extremely effective mathematical tool for specifying and verifying a vast class of synchronization and coordination problems common in concurrency. Pnueli thus caused a sea-change in the field of program verification, founding the time of reasoning about time, which has been the most successful period in formal methods yet.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, p. 1, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Static Verification for Code Contracts

Manuel Fähndrich

Microsoft Research
maf@microsoft.com

Abstract. The Code Contracts project [3] at Microsoft Research enables programmers on the .NET platform to author specifications in existing languages such as C# and VisualBasic. To take advantage of these specifications, we provide tools for documentation generation, runtime contract checking, and static contract verification.

This talk details the overall approach of the static contract checker and examines where and how we trade-off soundness in order to obtain a practical tool that works on a full-fledged object-oriented intermediate language such as the .NET Common Intermediate Language.

1 Code Contracts

Embedding a specification language in an existing language consists of using a set of static methods to express specifications inside the body of methods [4].

```
1  string TrimSuffix(string original, string suffix)
2  {
3    Contract.Requires(original != null);
4    Contract.Requires(!String.IsNullOrEmpty(suffix));
5
6    Contract.Ensures(Contract.Result() != null);
7    Contract.Ensures(!Contract.Result().EndsWith(suffix));
8
9    var result = original;
10   while (result.EndsWith(suffix)) {
11     result = result.Substring(0, result.Length - suffix.Length);
12   }
13   return result;
14 }
```

The code above specifies two preconditions using calls to `Contract.Requires` and two postconditions using calls to `Contract.Ensures`. These methods have no intrinsic effect and are just used as markers in the resulting compiled code to identify the preceeding instructions as pre- or postconditions.

2 Verification Steps

Our analysis operates on the compiled .NET Common Intermediate Language [2] produced by the standard C# compiler. The verification is completely modular

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 2–5, 2010.
© Springer-Verlag Berlin Heidelberg 2010

in that we analyze one method at a time, taking into account only the contracts of called methods. In our example, the contracts of called String methods are:

```
int Length {
  get { Contract.Ensures(Contract.Result<int>() >= 0); }
}

static bool IsNullOrEmpty(string str)
{
  Contract.Ensures( Contract.Result<bool>() == (str == null || str.Length == 0) );
}

bool EndsWith(string suffix)
{
  Contract.Requires( suffix != null);

  Contract.Ensures( ! Contract.Result<bool>() || value.Length <= this.Length);
}

string Substring(int startIndex, int length)
{
  Contract.Requires(0 <= startIndex);
  Contract.Requires(0 <= length);
  Contract.Requires( startIndex + length <= this.Length);

  Contract.Ensures(Contract.Result<string>() != null);
  Contract.Ensures(Contract.Result<string>().Length == length);
}
```

We factor the code to be analyzed into subroutines: one subroutine per method body, one subroutine for a method's preconditions, and one subroutine for a method's postconditions. The actual code to be analyzed is then formed by inserting calls to appropriate contract subroutines in the method body. In our example, we insert a subroutine call to TrimSuffix's precondition on entry of the method, and a subroutine call to its postcondition on all exits of the method. Additionally, at each method call-site, we insert a call to the precondition subroutine of the called method just prior to the actual call, and a call to the corresponding postcondition subroutine immediately following the call.

The actual contract calls to Contract.Requires or Contract.Ensures turn into either **assert** or **assume** statements depending on their context. Requires on entry of a method turn into **assume** and Ensures on exit of a method turn into **assert**. Conversely, at call-sites, Requires turn into **assert**, and Ensures turn into **assume**.

Conditional branches are expanded into non-deterministic branches with **assume** statements on the outgoing edges. Additional proof obligations for implicit correctness conditions in MSIL, such as null-dereference checks and array bound checks can be automatically inserted into the analyzed code as **assert**s based on user preference.

In this manner, all conditions are simply sequences of MSIL instructions, no different than ordinary method body code, and all assumptions are **assume** statements, and all proof-obligations are **assert** statements.

2.1 Heap Abstraction

Next, the code is transformed into a scalar program by abstracting away the heap. This is the step where we allow some assumptions and approximations that are not safe in general in order to obtain a practical analysis that does not over-burden the programmer. First, we assume that memory locations not explicitly aliased by the code under analysis are non-aliasing. This is clearly an optimistic assumption, but works very well in practice. Second, we guess the set of heap locations that are modified at call-sites (we don't require programmers to write heap modification clauses). Our guesses are often conservative, but may be optimistic if our non-aliasing assumptions are wrong. These assumptions allow us to compute a value numbering for all values accessed by the code, including heap accessing expressions. We also introduce names for uninterpreted functions marked as [Pure] by the programmer. This provides reasoning over abstract predicates. Finally, abstracting the heap also removes old-expressions in postconditions that refer to the state of an expression at the beginning of the method.

To compute the value numbering, we break the control flow of the analyzed code into maximal tree fragments. The root of each tree fragment is a join point (or the method entry point) and is connected by edges to predecessor leafs of other tree fragments.

The set of names used by the value numbering is unique in each tree fragment. Edges connecting tree leafs to tree roots contain a set of assignments effectively rebinding value names from one fragment to the names of the next. The resulting code is in mostly passive form, where each instruction simply relates a set of value names. This form is ideal for standard abstract interpretation based on numerical domains. The assignments on rebinding edges between tree fragments provide a way to transform abstract domain knowledge prior to the join from one set of value names to the next, so that the join can operate on a common set of value names. The rebindings act as a generalization of ϕ-nodes. In contrast to ϕ-nodes which provide a join for each value separately, our rebindings form a join for the entire state simultaneously, which is crucial to maintain relational properties.

2.2 Abstract Interpretation Fixpoints

On the scalar program, we compute abstract program invariants for each program point based on standard abstract interpretation fixpoint techniques [1]. Our motivation to use abstract interpretation rather than theorem proving techniques is to enable programmers to use static verification without requiring them to write loop invariants. It also provides control over cost/precision trade-offs. We use a variety of novel domains such as Pentagons, Disintervals, and Subpoly-

hedra [5] to deal with relations that arise in practice. We also lift these domains over sequences in order to deal with universally quantified properties.

For each **assert** statement in the code, we attempt to discharge the proof obligation using the computed fixpoint at that program point. If the abstract state is strong enough to imply the obligation, the obligation is discharged. Otherwise, we attempt to discharge it using an additional backward analysis.

2.3 Weakest Precondition Analysis

If the abstract state at an **assert** is too weak to imply the proof obligation, we transform the obligation using weakest preconditions into obligations for all predecessor program points and attempt to use the abstract state at those points to discharge them. This approach is good and handling disjunctive invariants which our abstract domains typically don't represent precisely. E.g., an **assert** after a join point may not be provable due to loss of precision at the join. However, the abstract states at the program points just prior to the join may be strong enough to discharge the obligation. This backwards analysis discharges an obligation if it can be discharged on all paths leading to the assertion. It thus acts as a form of on-demand trace partitioning.

3 Conclusion

For the example code, our verification discharges 5 implicit non-null obligations on the receiver of the calls to EndsWith, Substring, and Length. It also discharges all preconditions of these methods as well as the postconditions of TrimSuffix.

Our tools have been available to the general public since March 2009 and the response has been very positive. We received much useful feedback that has been incorporated back into the tools. We believe that our approach is viable and that we have made good progress towards our goal of enabling non-verification experts to start writing specifications and use tools to enforce better programming discipline. Still, much work remains to be done on the static verification with respect to better scalability, precision, and automation.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, ACM Press, New York (January 1977)
2. ECMA: Standard ECMA-355, Common Language Infrastructure (June 2006)
3. Fähndrich, M., Barnett, M., Logozzo, F.: Code Contracts (March 2009), http://research.microsoft.com/contracts
4. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. In: SAC 2010: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2103–2110. ACM Press, New York (2010)
5. Laviron, V., Logozzo, F.: Subpolyhedra: A (more) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)

Translation Validation of Loop Optimizations and Software Pipelining in the TVOC Framework

In Memory of Amir Pnueli

Benjamin Goldberg

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
goldberg@cs.nyu.edu

Abstract. Translation validation (TV) is the process of proving that the execution of a translator has generated an output that is a correct translation of the input. When applied to optimizing compilers, TV is used to prove that the generated target code is a correct translation of the source program being compiled. This is in contrast to verifying a compiler, i.e. ensuring that the compiler will generate correct target code for every possible source program – which is generally a far more difficult endeavor.

This paper reviews the TVOC framework developed by Amir Pnueli and his colleagues for translation validation for optimizing compilers, where the program being compiled undergoes substantional transformation for the purposes of optimization. The paper concludes with a discussion of how recent work on the TV of software pipelining by Tristan & Leroy can be incorporated into the TVOC framework.

1 Introduction

Verifying a compiler to ensure that it will produce correct target code every time it compiles a source program is a very difficult undertaking. First, compilers are large pieces of software and, given the current state of the art, verifying large pieces of software is still generally computationally intractable. Second, compilers tend to undergo updates and new releases, which would require re-verification each time.

As a proposed solution to the difficulty of verifying that a compiler will produce correct target code for any possible source program, starting in 1998 Amir Pnueli and his colleagues [10,9,11,8,12] proposed *translation validation* (TV), which is the process of verifying, for a given run of the compiler, that the target code produced during the run is a correct translation of the source program being compiled. Initially, the TV work was performed for a compiler that translated SIGNAL, a reactive language with very simple program structure (a single outer loop), into C. This work was followed up by Pnueli and various colleagues

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 6–21, 2010.
© Springer-Verlag Berlin Heidelberg 2010

(including this author), as well by many other researchers, who developed TV methods for industrial-strength optimizing compilers (see, e.g. [7,2,15,17] among too many to list).

Performing TV for optimizing compilers is especially challenging because the optimizations performed by the compiler can significantly change the structure of a program. The TV for optimizing compilers work performed by Pnueli and colleagues resulted in a framework and implementation called TVOC [2], for Translation Validation for Optimizing Compilers, which partitions compiler optimizations into two categories:

– *Structure-preserving optimizations*: These are optimizations that do not radically change the structure of the program, so that a mapping between states of the target program and states of the source program is still possible. Examples of such optimizations include the so-called "global optimizations", such as dead-code elimination, constant folding and propagation, and common subexpression elimination.
– *Structure-modifying optimizations*: These are optimizations that radically change the structure of a program – or at least parts of the program, such as loops – so that there is no useful mapping between states of the target program and states of the source program. Examples of these optimizations include loop optimizations such as loop interchange, tiling, reversal, fusion, and distribution.

These two categories are treated differently within the TVOC framework. In both cases, based on the source and target programs and the optimizations performed, the TVOC system generates verification conditions that are then checked by a theorem prover.

The theorem prover that TVOC uses is CVC [14,1], which is an automatic theorem prover for proving the validity of first-order formulas and has a large number of built-in theories that are useful for TV (e.g. integers, arrays, bit-vectors, etc.). The latest instantiation of CVC is CVC3 [1]. If CVC determines that the verification conditions that TVOC generates are satisfied, then the optimizations applied by the compiler were correct. Otherwise, the TVOC system indicates that the compilation was invalid.

Figure 1 shows a simple schematic of the TVOC system, as applied to the Intel Open Research Compiler a few years ago. After parsing and type checking (which the TVOC system does not validate), the compiler performs loop optimizations, global optimizations, and some machine dependent optimizations prior to code generation. Each optimization phase comprises one or more IR-to-IR transformations, taking the program in an intermediate reprentation (IR) and producing a new program represented in the same IR language. Based on these transformations, TVOC produces the set of verification conditions that are fed to the CVC theorem prover.

Figure 2 shows a slightly more detailed schematic of the TVOC system. There are separate components of TVOC for validating loop optimizations (structure modifying) and global optimizations (structure preserving). At this point, validation of machine-dependent optimizations has not been implemented in

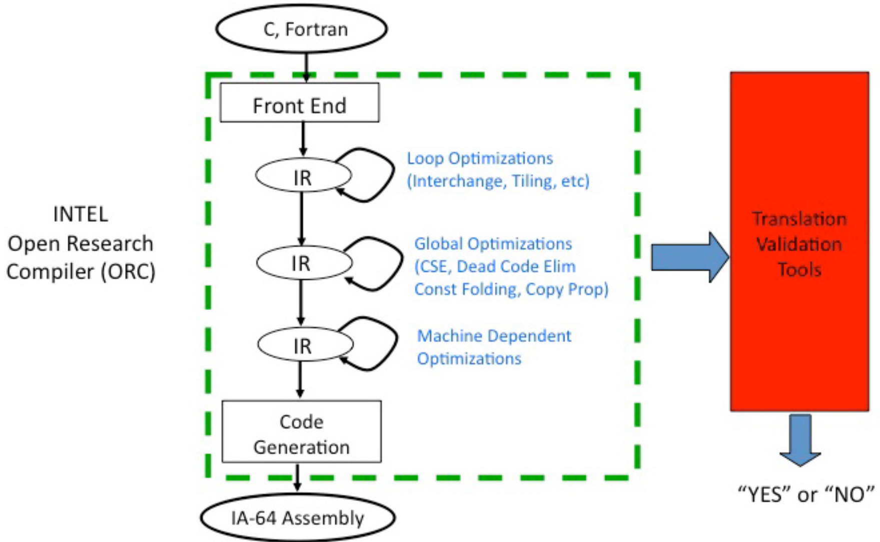


Fig. 1. A Simple Schematic of the TVOC System

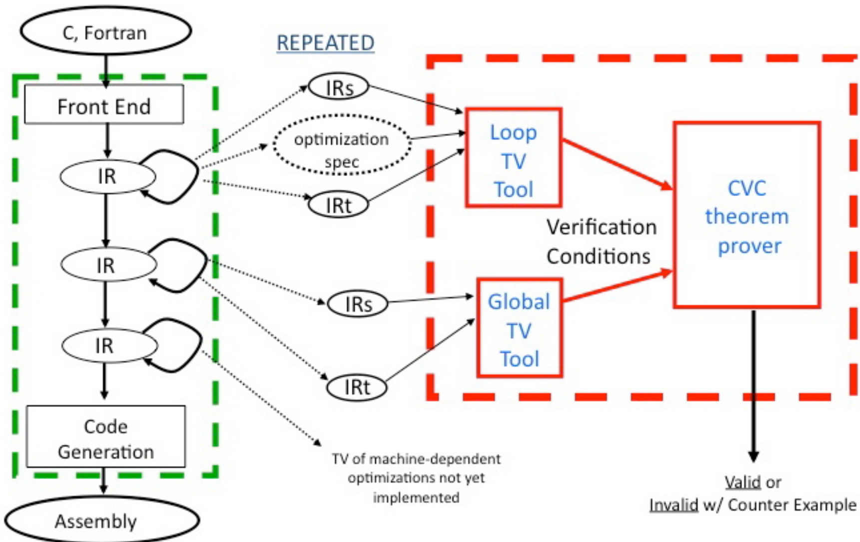


Fig. 2. Detailed Schematic of the TVOC System

TVOC. Loop optimizations are generally performed earlier in the compilation process than global optimizations, since loop optimizations often expose opportunities for global optimization. In any case, these optimization processes tend to be iterative. The input (source) and output (target) of each optimization is fed to the appropriate module (loop TV or global TV) of TVOC, which generates the verification conditions to be fed to CVC.

We have recently begun to extend the TVOC framework, although not the implementation yet, to handle machine-dependent optimizations. One such optimization that has not been handled by TVOC, although it was addressed in other Pnueli work, is software pipelining. Recent work by Tristan & Leroy [16] for validating software pipelining using symbolic evaluation is being been adapted for the TVOC framework (i.e. using CVC). We describe here how software pipelining fits into the TVOC framework.

2 Validating Global Optimizations in TVOC

Global optimizations are structure preserving in the sense that they preserve the structure of a program sufficiently to permit a mapping between states of the target program (i.e. the IR representation of the program after an optimization) and states of the source (i.e. the IR representation of the program before the optimization). Although a detailed explanation of how validation of global optimizations are performed in TVOC is beyond the scope of this paper, we provide a brief description here. We refer the reader to [18] for more details and examples.

In order to validate a translation from a source program S to a target program T, where the transformations applied to S are structure-preserving, TVOC represents each program as a *transition system* [10] (TS), which is a state machine consisting of a set V of state variables, a set $\mathcal{O} \subseteq V$ of observable variables, an initial condition Θ characterizing the initial states of the system, and a transition relation ρ relating each state to its possible successors. The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. A computation of a TS is defined to be a maximal finite or infinite sequence of states starting with a state that satisfies the initial condition such that every two consecutive states are related by the transition relation.

In order to establish that P_T, the TS representing the target program T, is a correct translation of P_S, the TS representing the source program S, we use a proof rule, Val, which is inspired by the computational induction approach [3], originally introduced for proving properties of a single program. Rule Val provides a proof methodology by which one can prove that one program *refines* another. This is achieved by establishing a *control mapping* from target to source locations, a *data abstraction* mapping from source variables to expressions over the target variables, and proving that these abstractions are maintained along basic execution paths of the target program.

In Val, each TS is assumed to have a *cut-point* set, i.e., a set of blocks that includes all initial and terminal blocks, as well as at least one block from each of the cycles in the programs' control flow graph. A *simple path* is a path connecting two cut-points, and containing no other cut-point as an intermediate node. For each simple path, we can (automatically) construct the transition relation of the path. Typically, such a transition relation contains the condition which enables this path to be traversed and the data transformation effected by the path.

Rule Val constructs a set of verification conditions, one for each simple target path, whose aggregate consists of an inductive proof of the correctness of the translation between source and target. Roughly speaking, each verification condition states that, if the target program can execute a simple path, starting with some conditions correlating the source and target programs, then at the end of the execution of the simple path, the conditions correlating the source and target programs still hold. The conditions consist of the control mapping, the data mapping, and, possibly, some invariant assertion holding at the target code.

3 Validating Loop Optimizations

The Val rule discussed above relied on there being a mapping between the states of the source and target programs. However, there is a class of loop optimizations that optimizing compilers perform that modify the structure of loops sufficiently so that no such mapping is possible. Thus, Pnueli and his colleagues, including this author, developed and implemented in TVOC a method for validating loop optimizations that did not rely on such a mapping. We describe this method briefly here, but refer the reader to [2].

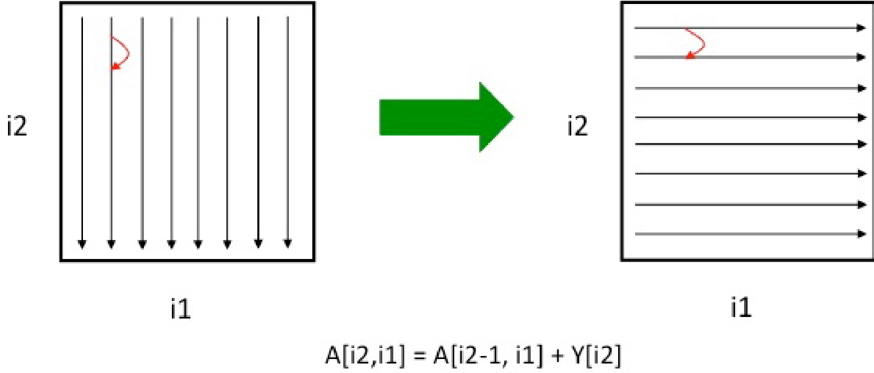
The loop optimizations that TVOC handles fall under the category of reordering transformations, which are transformations that change the order of execution of statements in the body of a loop, but do not change the number of times each statement is executed. Reordering transformations cover many of the loop optimizations performed by optimizing compilers, including fusion, distribution, reversal, interchange, and tiling.

To illustrate TVOC's validation of loop optimizations, we consider loop interchange. The loop interchange optimization reorders the nesting of a nested loop. Figure 3 shows an example of a loop interchange on a doubly-nested loop. For this example, the transformation may provide several performance benefits. First, since the expression `Y[i2]` is loop invariant in the inner loop of the transformed code, the computation of the address denoted by Y[i2] and the fetching of its value can be moved outside the inner loop. Second, if the array `A` is arranged in row major form, where adjacent elements in a row occupy consecutative locations in memory, then cache performance is likely to be improved. The illustration below shows the transformation of the access pattern over the array `A` caused by the interchange.

```
for i1 = 1 to N do
 for i2 = 2 to M do
   A[i2,i1] = A[i2-1, i1] + Y[i2]
 end
end
```

```
for i2 = 2 to M do
 for i1 = 1 to N do
   A[i2,i1] = A[i2-1, i1] + Y[i2]
 end
end
```

Fig. 3. An example of loop interchange



The curved arrows represent an execution of the assignment statement, where each element `A[i2,i1]` is assigned a value computed from the element `A[i2-1,i1]` above it. The new access pattern resulting from the optimization must preserve the relative order in which `A[i2,i1]` and `A[i2-1,i1]` are visited during execution of the loop. Otherwise, the transformation will have changed the result produced by the loop.

In order to define a single rule for validating all reordering loop transformations, we represent a loop of the form

$$\begin{array}{l} \textbf{for } i_1 = L_1 \textbf{to} H_1 \textbf{ do} \\ \quad \ldots \\ \qquad \textbf{for } i_m = L_m \textbf{ to } H_m \textbf{ do} \\ \qquad\qquad B(i_1, \ldots, i_m) \end{array}$$

by

$$\textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } B(\boldsymbol{i})$$

where $\boldsymbol{i} = (i_1, \ldots, i_m)$ is the list of nested loop indices, $\mathcal{I}$ is the set of the values assumed by $\boldsymbol{i}$ through the different iterations of the loop, and B represents the entire body of the loop. The set $\mathcal{I}$ can be characterized by a set of linear inequalities. For example, for the above loop, $\mathcal{I}$ is defined by

$$\mathcal{I} \quad = \quad \{(i_1, \ldots, i_m) \mid L_1 \leq i_1 \leq H_1 \;\wedge\; \cdots \;\wedge\; L_m \leq i_m \leq H_m\}$$

The relation $\prec_{\mathcal{I}}$ is the ordering by which the various points of $\mathcal{I}$ are traversed. For example, for the loop above, this ordering is the lexicographic order on $\mathcal{I}$.

In general, a loop transformation has the form:

$$\textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\boldsymbol{i}) \quad \Longrightarrow \quad \textbf{for } \boldsymbol{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\boldsymbol{j}))$$

Such a transformation may change the domain of the loop indices from $\mathcal{I}$ to $\mathcal{J}$, change the loop indices from $\boldsymbol{i}$ to $\boldsymbol{j}$, and possibly introduce an additional linear transformation in the loop's body, changing it from the source loop body $\mathtt{B}(\boldsymbol{i})$ to the target body $\mathtt{B}(F(\boldsymbol{j}))$.

The rule used in TVOC to validate loop transformations is the Permute rule shown in Figure 4, where F is a bijection (i.e. it is one-to-one and onto) mapping iterations in the transformed loop back to iterations in the original loop.

$$\frac{\forall \boldsymbol{i}_1, \boldsymbol{i}_2 \in \mathcal{I} : \boldsymbol{i}_1 \prec_{\mathcal{I}} \boldsymbol{i}_2 \wedge F^{-1}(\boldsymbol{i}_2) \prec_{\mathcal{J}} F^{-1}(\boldsymbol{i}_1) \implies \mathtt{B}(\boldsymbol{i}_1);\mathtt{B}(\boldsymbol{i}_2) \sim \mathtt{B}(\boldsymbol{i}_2);\mathtt{B}(\boldsymbol{i}_1)}{\textbf{for } \boldsymbol{i} \in \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathtt{B}(\boldsymbol{i}) \sim \textbf{for } \boldsymbol{j} \in \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathtt{B}(F(\boldsymbol{j}))}$$

Fig. 4. Permutation Rule Permute for reordering transformations

Intuitively, the Permute rule says that if, for any circumstance under which a reordering transformation switches the relative order of two iterations i_1 and i_2 in the source and target code, it is case that executing the body B in iteration i_1 followed by executing B in iteration i_2 is equivalent to executing B in iteration i_2 followed by executing the body in iteration i_1, then the reordering transformation is correct.

In order to apply rule Permute to a given case, it is necessary to identify the function F (and F^{-1}) and verify that the antecedent of Rule Permute is satisfied. The identification of F can be provided by the compiler, once it determines which of the relevant loop optimizations it chooses to apply. Intel's ORC compiler generates a file containing a description of the loop optimizations applied in the current phase of optimization. TVOC extracts this information (identified as "optimization spec" in Figure 2), verifies that the optimized code has resulted from the indicated optimization, and constructs the verification conditions. These conditions are then passed to CVC, which checks them automatically.

Consider the interchange example shown in Figure 3. The loop interchange transformation for that example can be characterized as follows:

$$\begin{array}{l}\textbf{for } \boldsymbol{i} \textbf{ in } \mathcal{I} \textbf{ by } \prec_{\mathcal{I}} \textbf{ do } \mathbf{A}[i_2 - 1, i_1] + \mathbf{Y}[i_2] \\ \Longrightarrow \\ \textbf{for } \boldsymbol{j} \textbf{ in } \mathcal{J} \textbf{ by } \prec_{\mathcal{J}} \textbf{ do } \mathbf{A}[j_1 - 1, j_2] + \mathbf{Y}[j_1]\end{array}$$

where

$$\mathcal{I} = \{(i_1, i_2) | 1 \leq i_1 \leq N, 1 \leq i_2 \leq M\}$$
$$\mathcal{J} = \{(j_1, j_2) | 1 \leq j_1 \leq M, 1 \leq j_2 \leq N\}$$

and $\prec_{\mathcal{I}}$ and $\prec_{\mathcal{J}}$ are lexicographic ordering on their respective iteration spaces. The functions F and F^{-1} associated with loop interchange are defined by

$$F(j_1, j_2) = (j_2, j_1)$$
$$F^{-1}(i_1, i_2) = (i_2, i_1)$$

In order to determine if loop interchange is valid on the example loop, the definitions of $\mathcal{I}$, $\mathcal{J}$, $\prec_{\mathcal{I}}$, $\prec_{\mathcal{J}}$,F, F^{-1}, and the loop body B are plugged into the antecedent of the Permute rule, namely

$$\forall \boldsymbol{i}_1, \boldsymbol{i}_2 \in \mathcal{I} : \boldsymbol{i}_1 \prec_{\mathcal{I}} \boldsymbol{i}_2 \wedge F^{-1}(\boldsymbol{i}_2) \prec_{\mathcal{J}} F^{-1}(\boldsymbol{i}_1) \implies \mathtt{B}(\boldsymbol{i}_1); \mathtt{B}(\boldsymbol{i}_2) \sim \mathtt{B}(\boldsymbol{i}_2); \mathtt{B}(\boldsymbol{i}_1)$$

The resulting formula is then fed to CVC to determine if it is valid. If it is valid, then loop interchange optimization is correct for this example.

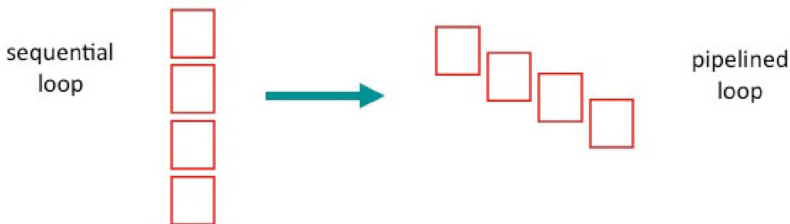
For those cases where the compiler does not indicate the loop transformations that were applied, TVOC uses a set of heuristics figure out which transformations were used.

4 Validating Software Pipelining

Machine-dependent optimizations, such as software pipelining, are not yet handled by the TVOC implementation. In this section, we discuss how TV for software pipelining can be incorporated into TVOC, based on recent work by Tristan & Leroy [16]. We start, however, with a intuitive explanation of the software pipelining optimization.

4.1 A Gentle Introduction to Software Pipelining

Software pipelining [13,5] refers to a class of optimizations that improve program performance by overlaying iterations of a loop – essentially allowing an iteration to start before the previous iteration has completed, even if there are dependences between iterations that prohibit the iterations executing fully in parallel. Software pipelining can be view schematically as:



The benefits of software pipelining include 1) exploiting instruction-level parallelism by allowing instructions from different iterations to execute simultaneously on VLIW or superscalar machines, 2) filling delay slots in one iteration with instructions from other iterations, and 3) other improvements (register allocation, cache performance, etc.) that can be made during instruction scheduling by being able to select among instructions from several overlapping iterations.

Although software pipelining generally occurs at the instruction-scheduling phase of compilation, where the optimization is applied to machine instructions, for clarity we will show the examples in this paper in an intermediate representation (IR) that is fairly close to the source.

Consider the following simple loop:

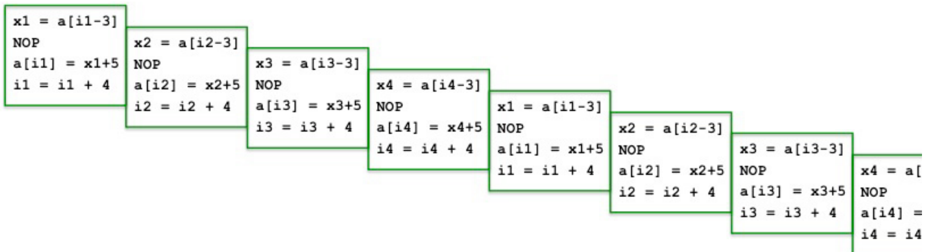
```
for i= 3 to N
  a[i] = a[i-3] + 5
```

A corresponding (high-level) intermediate representation form of the loop is:

```
i=3
while (i<=N) {
   x = a[i-3]
   NOP //delay slot
   a[i] = x+5
   i = i + 1
   }
```

We assume that the load instruction, `x = a[i-3]`, takes an extra cycle due to the memory fetch, thus a NOP ("no-op") is inserted to ensure that `x` is not referenced too early[1]. In the sequential execution of the loop, a cycle is wasted by the NOP during every iteration.

The figure below illustrates the execution of overlaid iterations in a software pipeline. These iterations continue executing as long as specified by the loop bounds.



As can be seen by close examination of the above figure, the actual pipeline code is accomplished by replicating the body of the loop four times, creatng a total of four instances of the variables `i` and `x`, and then overlaying the four iterations.

[1] For simplicity, we assume a purely statically-scheduled machine with no out-of-order execution or interlocked stages.

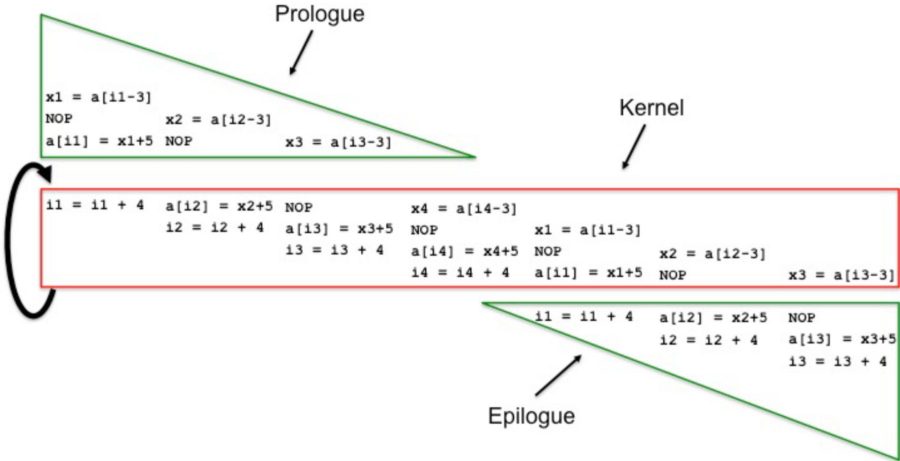
During execution, these four iterations are repeatedly executed, as implied by the figure above.

In a software pipeline, such as the one illustrated above, the instructions appearing on the same horizontal level – despite being from different iterations – can be executed simultaneously or in any order chosen by the compiler. Thus, although the NOP appears in the figure, it does not consume a cycle since there are other instructions that can be executed in that same cycle.

Upon further examination of the above figure, it can be seen that horizontal blocks of code are repeated in the execution of the overlaid iterations. This is shown in the figure below, where the code within the first large rectangle is repeated in the second rectangle (which is only partially visible) and many times subsequently.

```
x1 = a[i1-3]
NOP            x2 = a[i2-3]
a[i1] = x1+5   NOP            x3 = a[i3-3]
i1 = i1 + 4    a[i2] = x2+5   NOP            x4 = a[i4-3]
               i2 = i2 + 4    a[i3] = x3+5   NOP            x1 = a[i1-3]
                              i3 = i3 + 4    a[i4] = x4+5   NOP            x2 = a[i2-3]
                                             i4 = i4 + 4    a[i1] = x1+5   NOP            x3 = a[i3-3]
                                                            i1 = i1 + 4    a[i2] = x2+5   NOP            x4 = a[i4-3
                                                                           i2 = i2 + 4    a[i3] = x3+5   NOP
                                                                                          i3 = i3 + 4    a[i4] = x4+
                                                                                                         i4 = i4 + 4
```

The horizontal block of code within the large rectangle is called the "kernel" of the pipeline. Only one instance of the kernel code is actually generated, and is then executed in a loop. The figure below illustrates the repeated execution of the kernel code, preceded by a set of instructions called the "prologue" and followed by the set of instructions called the "epilogue". The prologue can be thought of as a "ramping up" of the pipeline and the epilogue as a "ramping down" of the pipeline.



For clarity, the above figure doesn't show the number of times that the kernel is executed. It can be seen from inspection that, together, the prologue and

epilogue corresponds to executing three iterations of the original loop body (note the three assignments to `x`, the three writes to `a[ ]`, etc.) and that the kernel code corresponds to four iterations of the original loop body. Thus, since the original loop executed N times, it must be the case that N is at least 3, since the prologue and epilogue will always execute once the pipelined code is entered. Furthermore, the value of $N-3$, i.e. the number of iterations of the original loop that is executed by iterating over the kernel, must be divisible by four since each iteration of the kernel corresponds to four iterations of the original loop.

Using this logic, and the notation from [16], it is clear that, in general, if the prologue and epilogue together execute μ iterations of the original loop and each iteration of the kernel executes δ iterations of the original loop, then we require that $N \geq \mu$ and that $(N - \mu)$ is a multiple of δ. To enforce these requirements, the pipeline code is generally preceded by a conditional that tests the value of N, unless N can be determined statically. If $N < \mu$, then the pipeline code will not be entered at all. If $N-\mu$ is not a multiple of δ, then the appropriate number (i.e. $(N-\mu)$ MOD δ) of iterations of the loop are peeled off and executed separately, so that the remaining iterations of the loop can be pipelined.

4.2 Validating a Software Pipeline

In [6], Pnueli and Leviathan described a method for validating software pipelining using an extension of the Val rule described above. This work used a mapping between transition systems resulting in a fairly complicated method.

In a recent POPL paper [16], Tristan & Leroy describe a less complicated approach, defining a simple rule to be satisfied in order to deem that the translation from the original loop into a pipeline is correct. As their paper discusses, given a source loop with a body B that is translated into the pipeline consisting of a prologue P, a kernel S, and an epilogue E, where E and P together represent μ iterations of B and S represents δ iterations of B, the translation is correct *iff*

$$B^N \sim P; S^{(N-\mu)/\delta}; E$$

That is, executing the body B of a loop N times is equivalent to executing the prologue P, followed by iterating over the kernel S for $(N - \mu)/\delta$ times, followed by the epilogue E. As discussed above, it is assumed (and enforced by other code) that $N \geq \mu$ and that $(N - \mu)$ is a multiple of δ. Tristan & Leroy noted, though, that without knowledge of N, which is a run-time value, proving the above equivalence for all possible N is very difficult. Thus, they proposed a simple rule that is sound but not complete, in that if the rule is satisfied, then the translation is correct, but there may be correct translations that do not satisfy the rule. However, their paper states that such cases don't arise in practice.

The Tristan & Leory rule can be specified as follows: Suppose a source loop whose body is B is translated into the pipeline consisting of a prologue P, a kernel S, and an epilogue E, where E and P together represent μ iterations of B and S represents δ iterations of B. Then,

$$\frac{(B^{\mu} \sim P; E) \wedge (E; B^{\delta} \sim S; E)}{B^{N} \sim P; S^{(N-\mu)/\delta}; E} \qquad \text{(Tristan \& Leroy)}$$

where it is assumed that $N \geq \mu$ and $(N - \mu)$ is a multiple of δ.

As shown in their POPL paper, the Tristan & Leroy rule is easy to prove inductively (once a framework, such as their symbolic evaluation, is developed for reasoning about equivalence – which is not so easy). Informally, the induction proceeds as follows. Since $N - \mu$ is divisble by δ, $N = \mu + m\delta$ for some $m \geq 0$. m is used as the basis of the induction.

Base Case $m = 0$:

$$\begin{aligned} B^{\mu+m\delta} &= B^{\mu} \\ &\sim P; E \\ &= P; S^{0}; E \end{aligned}$$

Assume for any $m \leq k$, $B^{\mu+k\delta} \sim P; S^{k}; E$. Then,

$$\begin{aligned} B^{\mu+(k+1)\delta} &= B^{\mu+k\delta}; B^{\delta} \\ &\sim P; S^{k}; E; B^{\delta} \\ &\sim P; S^{k}; S; E \\ &= P; S^{k+1}; E \end{aligned}$$

Thus, for any m, $B^{\mu+m\delta} \sim P; S^{m}; E$ and since $N = \mu + m\delta$, i.e. $m = (N - u)/\delta$, $B^{N} \sim P; S^{(N-\mu)/\delta}; E$.

The intuition behind the Tristan & Leroy rule can be seen in figure 5, which is adapted (with permission) from Figure 3 in [16]. The horizontal sequence at the top of the figure represents the execution of the original code and the sequence at the bottom is the execution of the pipelined code.

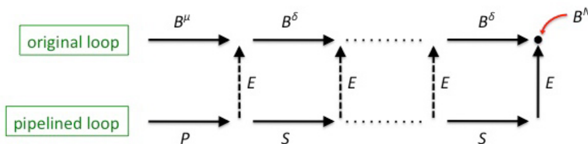


Fig. 5. Illustration of the Tristan & Leroy rule, adapted from [16]

In their POPL paper, Tristan & Leroy describe a symbolic evaluation method for proving the equivalences ($B^\mu \sim P; E$) and ($E; B^\delta \sim S; E$) for a particular source loop body B and target pipeline components P, S, and E. Instead, we have incorporated the Tristan & Leroy rule into the TVOC framework, where it is used to generate two verification conditions – simply ($B^\mu \sim P; E$) and ($E; B^\delta \sim S; E$) – that are fed to CVC theorem prover, along with the code for B^μ, B^δ, P, S, and E. If CVC finds the two conditions valid, then pipelining is correct.

Figure 6 shows the original and pipelined loops of our example program, above, along with the verification conditions, encoded for CVC. P, S, and E in the CVC code resulted from an SSA transformation applied to the pipeline code. B^3 and B^4, corresponding to B^μ, B^δ, respectively, were generated by static loop unrolling and then an SSA transformation. Equivalence between B^3 and $P; E$ and between $E; B^4$ and $S; E$ is checked in CVC by asserting that their inputs (the initial values of `a`, the `i`'s, and the `x`'s) are equal and querying CVC about the equality of the their outputs (i.e. the final values of `a`, the `i`'s, and the `x`'s).

Software pipelining, although an optimization that can be complicated to perform, lends itself nicely to simple translation validation rules, such as the Tristan & Leroy rule, because none of the pipeline prologue, kernel, or epilogue themselves contain loops or branches. Although the compiler has freedom to rearrange instructions within each of these blocks, the resulting code will still be amenable to equivalence checking by a theorem prover.

4.3 Future Work: Validating Pipelining That Uses Hardware Support

In practice, compilers that perform software pipelining often generate code for machines, such as the Intel IA64, that provides substantial hardware support for pipelining. This hardware support includes rotating registers to provide automatic renaming of variables (such as the loop index `i` in our example above) across iterations – thus avoiding replicating identical code in overlapping iterations and reducing the size of the kernel code. Another form of hardware support for software pipelining is predication, which is the ability to turn off the execution of certain instructions at run time. Predication, in this case, supports the execution of prologue and epilogue code – which are subsets of the kernel instructions – by turning off certain instructions in the kernel during the ramp up and ramp down phases of the pipeline. As described in [4], predication can also be used to dynamically alter the software pipeline in order to preserve loop-carried dependences that can only be computed at run time.

Techniques for translation validation of software pipelining that use such hardware support have not yet been developed. As with performing TV for other kinds of machine-dependent optimizations, it will involve encoding the hardware features of the machine in a logical framework (e.g. as a set of CVC assertions).

Unrolled Source Code and Target Pipline Code

```
%B3
REAL_ARRAY: TYPE = ARRAY INT OF REAL;
a1_b3: REAL_ARRAY;
x1_b3: REAL = a1_b3[i1_b3-3];
a2_b3: REAL_ARRAY = a1_b3 WITH [i1_b3] := x1_b3 + 5;
x2_b3: REAL = a2_b3[i1_b3-2];
a3_b3: REAL_ARRAY = a2_b3 WITH [i1_b3 + 1] := x2_b3 + 5;
x3_b3: REAL = a3_b3[i1_b3-1];
a4_b3: REAL_ARRAY = a3_b3 WITH [i1_b3 + 2] := x3_b3 + 5;
i2_b3: INT = i1_b3 + 3;
%B4
a1_b4: REAL_ARRAY;
i1_b4: INT;
x1_b4: REAL = a1_b4[i1_b4-3];
a2_b4: REAL_ARRAY = a1_b4 WITH [i1_b4] := x1_b4 + 5;
x2_b4: REAL = a2_b4[i1_b4-2];
a3_b4: REAL_ARRAY = a2_b4 WITH [i1_b4 + 1] := x2_b4 + 5;
x3_b4: REAL = a3_b4[i1_b4-1];
a4_b4: REAL_ARRAY = a3_b4 WITH [i1_b4 + 2] := x3_b4 + 5;
x4_b4: REAL = a4_b4[i1_b4];
a5_b4: REAL_ARRAY = a4_b4 WITH [i1_b4 + 3] := x4_b4 + 5;
i2_b4: INT = i1_b4 + 4;
%PROLOGUE
i1_pl: INT;
i2_pl: INT;
i3_pl: INT;
i4_pl: INT;
ASSERT i1_pl = 2;
ASSERT i2_pl = i1_pl + 1;
ASSERT i3_pl = i1_pl + 2;
ASSERT i4_pl = i1_pl + 3;
a1_pl: REAL_ARRAY;
x1_pl: REAL = a1_pl[i1_pl-3];
x2_pl: REAL = a1_pl[i2_pl-3];
a2_pl: REAL_ARRAY = a1_pl WITH [i1_pl] := x1_pl + 5;
x3_pl: REAL = a2_pl[i3_pl-3];
%EPILOGUE
a1_ep: REAL_ARRAY;
i11_ep: INT;
i21_ep: INT;
i31_ep: INT;
i41_ep: INT;
x1_ep: REAL;
x2_ep: REAL;
x3_ep: REAL;
x4_ep: REAL;
i12_ep: INT = i11_ep + 4;
a2_ep: REAL_ARRAY = a1_ep WITH [i21_ep] := x2_ep + 5;
i22_ep: INT = i21_ep + 4;
a3_ep: REAL_ARRAY = a2_ep WITH [i31_ep] := x3_ep + 5;
i32_ep: INT = i31_ep + 4;
%EPILOGUE2, a copy of EPILOGUE
a1_ep2: REAL_ARRAY;
i11_ep2: INT;
i21_ep2: INT;
i31_ep2: INT;
i41_ep2: INT;
x2_ep2: REAL;
x3_ep2: REAL;
i12_ep2: INT = i11_ep2 + 4;
a2_ep2: REAL_ARRAY = a1_ep2 WITH [i21_ep2] := x2_ep2 + 5;
i22_ep2: INT = i21_ep2 + 4;
a3_ep2: REAL_ARRAY = a2_ep2 WITH [i31_ep2] := x3_ep2 + 5;
i32_ep2: INT = i31_ep2 + 4;
%KERNEL (S)
a1_s: REAL_ARRAY;
i11_s: INT;
i21_s: INT;
i31_s: INT;
i41_s: INT;
x21_s: REAL;
x31_s: REAL;
i12_s: INT = i11_s + 4;
a2_s: REAL_ARRAY = a1_s WITH [i21_s] := x21_s + 5;
x41_s: REAL = a2_s[i41_s - 3];
i22_s: INT = i21_s + 4;
a3_s: REAL_ARRAY = a2_s WITH [i31_s] := x31_s + 5;
x11_s: REAL = a3_s[i12_s - 3];
i32_s: INT = i31_s + 4;
a4_s: REAL_ARRAY = a3_s WITH [i41_s] := x41_s + 5;
x22_s: REAL = a4_s[i22_s - 3];
i42_s: INT = i41_s + 4;
a5_s: REAL_ARRAY = a4_s WITH [i12_s] := x11_s + 5;
x32_s: REAL = a5_s[i32_s - 3];
```

Assertions and Queries for Validation

```
%Assertions for P;E
%Connect the outputs of P to the
i1_b3: INT; %inputs of E.
ASSERT i11_ep = i1_pl;
ASSERT i21_ep = i2_pl;
ASSERT i31_ep = i3_pl;
ASSERT i41_ep = i4_pl;
ASSERT x2_ep = x2_pl;
ASSERT x3_ep = x3_pl;
ASSERT a1_ep = a2_pl;
%QUERIES FOR B3 = P;E
%Set the inputs to B3̂ equal to inputs to P
ASSERT a1_b3 = a1_pl;
ASSERT i1_b3 = i1_pl;

%Query if the outputs of B3 and E are equal
QUERY i2_b3 = i41_ep;
QUERY a3_ep = a4_b3;
QUERY x3_b3 = x3_ep;

%———————————-
%Assertions for S;E
%For S;E, the i1s, i2s, xs, and a's have to align
ASSERT i11_ep = i12_s;
ASSERT i21_ep = i22_s;
ASSERT i31_ep = i32_s;
ASSERT i41_ep = i42_s;
ASSERT x1_ep = x11_s;
ASSERT x2_ep = x22_s;
ASSERT x3_ep = x32_s;
ASSERT x4_ep = x41_s;
ASSERT a1_ep = a5_s;
%Assertions for E2;B4
%Need to use second copy of E, namely "_ep2"
%Align a[] output of E2 with a[] input of B4
ASSERT a1_b4 = a3_ep2;
%Align i1 input of B4 with i4 output of E2
ASSERT i1_b4 = i41_ep2;
%Queries for E2;B4 = S;E
%Assert the equality of the inputs to E2 and S
ASSERT a1_ep2 = a1_s;
ASSERT i11_ep2 = i11_s;
ASSERT i21_ep2 = i21_s;
ASSERT i31_ep2 = i31_s;
ASSERT i41_ep2 = i41_s;
ASSERT x2_ep2 = x21_s;
ASSERT x3_ep2 = x31_s;
%This gives the relationship among the i's in S
ASSERT i21_s = i11_s + 1;
ASSERT i31_s = i11_s + 2;
ASSERT i41_s = i11_s + 3;
ASSERT x1_ep = x11_s;
%Query if the outputs of B4 and E are equal.
QUERY i41_ep = i2_b4;
QUERY i12_ep = i2_b4+1;
QUERY i22_ep = i2_b4 + 2;
QUERY i32_ep = i2_b4 + 3;
QUERY x1_b4 = x4_ep;
QUERY x2_b4 = x1_ep;
QUERY x3_b4 = x2_ep;
QUERY x4_b4 = x3_ep;
QUERY a5_b4 = a3_ep;
```

Fig. 6. The pipelining example in CVC

5 Conclusion

We have attempted in this paper to provide an inkling of the contribution that Amir Pnueli made to techniques for ensuring the correctness of compilers – and the extent to which his translation validation work has inspired further work in this area. A large number of papers (too many to list here, unfortunately) have been published on translation validation since Pneuli's 1998 paper, and we expect translation validation to be an important area of verification for some time.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005)
3. Floyd, R.W.: Assigning meanings to programs. In: Proc. Symp. Appl. Math., vol. 19, pp. 19–31 (1967)
4. Goldberg, B., Crutcher, E., Huneycutt, C., Palem, K.: Software bubbles: Using predication to compensate for aliasing in software pipelines. In: International Conference on Parallel Architectures and Compilation Techniques, p. 211 (2002)
5. Lam, M.: Software pipelining: an effective scheduling technique for vliw machines. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI 1988), pp. 318–328 (July 1988)
6. Leviathan, R., Pnueli, A.: Validating software pipelining optimizations. In: CASES 2002: Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 280–287. ACM Press, New York (2002)
7. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI 2000: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 83–94. ACM Press, New York (2000)
8. Pnueli, A., Shtrichman, O., Siegel, M.: The code validation tool CVT: Automatic verification of a compilation process. International Journal on Software Tools for Technology Transfer (STTT) 2(1), 192–201 (1998)
9. Pnueli, A., Shtrichman, O., Siegel, M.: Translation validation for synchronous languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 235–246. Springer, Heidelberg (1998)
10. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
11. Pnueli, A., Shtrichman, O., Siegel, M.: Translation validation: From DC+ to C*. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 137–150. Springer, Heidelberg (1999)
12. Pnueli, A., Strichman, O., Siegel, M.: Translation validation: From SIGNAL to C. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 231–255. Springer, Heidelberg (1999)
13. Rau, B.R., Glaeser, C.D.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: MICRO 14: Proceedings of the 14th Annual Workshop on Microprogramming, Piscataway, NJ, USA, pp. 183–198. IEEE Press, Los Alamitos (1981)

14. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A cooperating validity checker. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 500–504. Springer, Heidelberg (2002)
15. Tristan, J.-B., Leroy, X.: Verified validation of lazy code motion. In: PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 316–326. ACM Press, New York (2009)
16. Tristan, J.-B., Leroy, X.: A simple, verified validator for software pipelining. In: POPL 2010: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 83–92. ACM, New York (2010)
17. Zaks, A., Pnueli, A.: CovaC: Compiler validation by program analysis of the cross-product. In: Cuéllar, J., Maibaum, T.S.E., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008)
18. Zuck, L.D., Pnueli, A., Goldberg, B.: VOC: A methodology for the translation validation of optimizing compilers. J. UCS 9(3), 223–247 (2003)

Size-Change Termination and Transition Invariants

Matthias Heizmann[1], Neil D. Jones[2], and Andreas Podelski[1]

[1] University of Freiburg, Germany
[2] University of Copenhagen, Denmark

Abstract. Two directions of recent work on program termination use the concepts of size-change termination resp. transition invariants. The difference in the setting has as consequence the inherent incomparability of the analysis and verification methods that result from this work. Yet, in order to facilitate the crossover of ideas and techniques in further developments, it seems interesting to identify which aspects in the respective formal foundation are related. This paper presents initial results in this direction.

1 Introduction

There have been rapid advances in methods for automatically proving program termination in recent years, both in theoretical research and in applications as practical as finding termination bugs in device drivers. A recent wave of activity began with the work on *size-change termination* from [27]. Related work and further developments include, e.g., [7,24,27,36]). A branch of this work is based on the concept of *transition invariants* from [31]; see, e.g., [11,14,15,18,26,32]). The motivation behind the work in [31] was to carry over the ideas of [27] to verification methods in the style of *software model checking* [3,4]. This goal entailed going from a *decidable* program analysis problem (for functional programs) to an *undecidable* verification problem (for imperative and concurrent programs). The change of setting has as consequence the inherent *incomparability* of the methods that result from the work on size-change termination resp. transition invariants. Yet, in order to facilitate the crossover of ideas and techniques in further developments of such methods, it seems interesting to identify which aspects in the respective formal foundation are related. This paper presents three initial results. They concern 1. the soundness proof, 2. the abstract domain, and 3. the base algorithm.

1. If we take the proof rule that implicitely underlies the soundness proof for the size-change termination analysis in [27] and the transition invariant-based proof rule from [31], then the premise of the former is strictly stronger than the premise of the latter and the conclusion of the former is strictly stronger than the conclusion of the latter (i.e., no proof rule subsumes the other one).

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 22–50, 2010.
© Springer-Verlag Berlin Heidelberg 2010

In detail: The size-change termination analysis in [27] decides *size-change termination*, a property strictly stronger than termination. The intermediate result of the analysis is a set of *size-change graphs*. The analysis gives a yes-answer if *some* of the graphs (the idempotent ones) denote a well-founded relation. But then, perhaps surprisingly, *all* of the graphs must denote a well-founded relation. This means that the premise in the (complete) proof rule for termination from [31] is satisfied.

2. The abstract domain of *size-change graphs* in [27] corresponds to a specific parameter for the *transition predicate abstraction* used in [11,14,15,18,32]. In detail: we can fix a specific set of *transition predicates* such that each size-change graph can be translated to an equivalent conjunction of transition predicates in this set, and vice versa. In fact, the arcs correspond to the conjuncts.
3. When we categorize the base algorithm in the termination analyses by the decision problem that it solves, we can establish the formal connection between the base algorithms.

   In detail: We define two decision problems, one for size-change termination and one for transition invariants; let us call them SCT and TI, for short. Then SCT belongs to a special case of a third decision problem which, in the special case, can be formulated in terms of TI (in general, the third decision problem has a strictly higher complexity than TI).

The special case of the third decision problem (in Point 3.) is defined by the associativity of the *abstract composition* of relations. The associativity is responsible not only for the lower complexity but also for the already (in Point 1.) mentioned feature of size-change termination analysis. I.e., among the elements in the output of the base algorithm, only the subset of *idempotent* elements has to be inspected for well-foundedness (if the binary operation over the elements is associative).

The definition of the special case thus abstracts away from the graph representation and helps us to identify the associativity of their composition as the crucial property of size-change graphs.[1]

The question left open by this paper is whether the notions of associativity and idempotency have correspondent notions for a similar optimization in transition invariant-based termination analyses.

Roadmap. The first part of this paper presents what we believe is the essence of size-change termination (Section 2) and transition invariants and transition predicate abstraction (Section 3). Points 1. and 2. from above are covered in Section 4. The reader who in interested only in Point 3. can jump directly to Section 5, which we tried to keep self-contained. The paper ends with a discussion of the qualitative differences that result from the different settings of the methods.

[1] The associativity of the composition is lost in the extension of size-change graphs with finitely many arc weights in the style of [5] (where, for example, the weighted arc $\mathtt{x} \xrightarrow{k} \mathtt{x}$ means that the value of $\mathtt{x}$ decreases by at least the integer k).

2 Size-Change Termination (SCT)

2.1 A Running Example

Example 1. Figure 1 is an program example similar to one in [24]. It is a first-order tail-recursive functional program with three function calls labeled 1, 2 and 3. Argument values range over the natural numbers $I\!N$, ordered as usual.

Figure 2 contains the program's "control flow graph" with the calling function and called function of each call, e.g., $\texttt{1} : \texttt{f} \to \texttt{g}$. It also associates with with each call τ a "size-change graph", e.g., G_τ. Example: G_1 abstracts the tuple of data flow size changes that occur in call 1 from f to g. Symbol $\downarrow$ in G_1, G_2, G_3 indicates a value decrease, and symbol $\overset{=}{\downarrow}$ indicates a decrease or equality.

```
f(x,y)   = if x=0 then y else 1: g(x,y,y)
g(u,v,w) = if v>0 then 2: g(u,v-1,2*w) else 3: f(u-1,w)
```

Fig. 1. Example of a first order tail-recursive functional program

Informal SCT Termination Reasoning for the Running Example. Suppose (hypothetically) there is *an infinite call sequence* $\pi = \tau_1\tau_2\tau_3\ldots$ that follows program P's control flow. We argue that any computation following π would have an infinitely descending sequence of variable values. But this would contradict the well-foundedness of set $I\!N$. Conclusion: program P terminates.

Case 1: $\pi = \ldots 2^\omega$ ends in infinitely many 2's. By safety of graph G_2, this implies that the values of *variable* v *descend infinitely.*

Case 2: Since π is infinite, the only other possibility is that it has the form $\pi = \ldots(12^*3)^\omega$. Again by safety, this implies that the values of *variable* u *descend infinitely* (once each time loop 12^*3 is traversed).

Therefore a call of *any program function with any data will terminate.*

Paper [27] shows two different approaches to make such reasoning algorithmic: One is based on Büchi automata, and the other computes the *closure* of the given set of graphs as follows (Section 1.2 of [27], and Section 2.4 below).

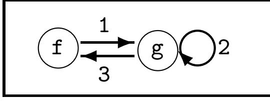
2.2 Some Size-Change Definitions

Program semantics: [27] is about first-order functional programs, and contains both syntax and a denotational (big-step) call-by-value semantics. Given a set *Value* containing values of expressions, the semantic function has type

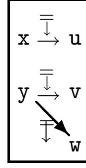
$$\mathcal{E}[\![_]\!] : \mathit{Expression} \to (\mathit{Value}^n \to \mathit{Value} \cup \{\bot\})$$

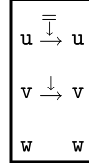
If e is an expression, then $\mathcal{E}[\![\texttt{e}]\!]\boldsymbol{v}$ is the value of expression e, given an an environment $\boldsymbol{v}$ containing values of the variables occurring in e. We omit the completely standard definition of $\mathcal{E}[\![_]\!]$, see [27] or a textbook on semantics for details.

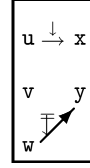
Control flow graph



Size-change graph set $\mathcal{G}$







$G_1 : \mathtt{f} \to \mathtt{g}$ $G_2 : \mathtt{g} \to \mathtt{g}$ $G_3 : \mathtt{g} \to \mathtt{f}$

Fig. 2. Size-change graphs for the running example

Size changes: we assume given a well-founded order $>$ on *Value*.

Definition 2. *Suppose functions* f, g *are defined in* P. *A* size-change graph $G : \mathtt{f} \to \mathtt{g}$ *for* P *is a set of labeled arcs* $x \xrightarrow{r} y$ *where* $r \in \{\bar{\downarrow}, \downarrow\}$, $x \in \mathsf{V}ariables(\mathtt{f})$, $y \in \mathsf{V}ariables(\mathtt{g})$, *and* G *does not contain both* $x \xrightarrow{\bar{\downarrow}} y$ *and* $x \xrightarrow{\downarrow} y$ *for any* x, y.

Functions f and g are respectively called the *source* and the *target* of G. We will sometimes elide f and g, writing G rather than $G : \mathtt{f} \to \mathtt{g}$.

Definition 3. *Let* $\mathcal{G} = \{G_\tau \mid \tau \text{ is a call in } P\}$ *be a set of size-change graphs for program* P.

1. *Suppose the definition of* f *contains a call to* g *labeled* τ:

$$\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_m) = \ldots \tau : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \ldots$$

   The phrase "arc $\mathtt{f}^{(i)} \xrightarrow{r} \mathtt{g}^{(j)}$ *safely describes the* $\mathtt{f}^{(i)}$-$\mathtt{g}^{(j)}$ *size relation in call* τ*" means: For every* $v \in$ *Value and* $\boldsymbol{v} = (v_1, \ldots, v_m)$, *if* $\mathcal{E}[\![\mathtt{e}_j]\!]\boldsymbol{v} = v$ *is defined, then*

$$r = \downarrow \text{ implies } v_i > v \text{ ; and } r = \bar{\downarrow} \text{ implies } v_i \geq v$$

2. *Size-change graph* G_τ *is* safe for call $\tau : \mathtt{f} \to \mathtt{g}$ *if every arc in* G_τ *is a safe description as just defined.*
3. *Set* $\mathcal{G}$ *of size-change graphs is a* safe description of program P *if graph* G_τ *is safe for every call* τ.

Assuming values are natural numbers, it is easy to see that all the size-change graphs shown example 1 are safe for their respective calls. No size relation in $\{\bar{\downarrow}, \downarrow\}$ can be safely asserted about argument w of call 2, since 2*w may exceed the current value of w. According to Definition 3, G_2 safely models the parameter size-changes caused by call 2.

Definition 4. *A* multipath $\mathcal{M}$ *is a graph sequence* $G_1, G_2, G_3, \ldots$ *such that* $\text{target}(G_i) = \text{source}(G_{i+1})$ *for* $i = 1, 2, \ldots$ *A* thread *is a connected path of arcs in* $\mathcal{M}$ *that starts at some* G_t, $t \geq 1$: $th = z_{i_t} \xrightarrow{r_t} z_{i_{t+1}} \xrightarrow{r_{t+1}} z_{i_{t+2}} \xrightarrow{r_{t+2}} \ldots$ *with each* $r_{t+j} \in \{\bar{\downarrow}, \downarrow\}$. *The thread has* infinite descent *if it contains infinitely many* $\downarrow$*'s.*

For example, G_2, G_3, G_1 is a multipath in Figure 2. It contains one thread with 3 arcs, namely $\mathtt{u} \xrightarrow{\overset{=}{\downarrow}} \mathtt{u} \xrightarrow{\downarrow} \mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{u}$.

Definition 5 (Size-change terminating program). *(Section 1.2 of [27]) Let $\mathcal{T}$ be the set of calls in program P. Suppose each size-change graph $G_\tau : \mathtt{f} \to \mathtt{g}$ is safe for every call τ in*

$$\mathcal{G} = \{G_\tau \mid \tau \in \mathcal{T}\}$$

Define P to be size-change terminating *if, for any infinite call sequence $\pi = \tau_1\tau_2\tau_3\ldots$ that follows P's control flow, there is a thread of infinite descent in the multipath $\mathcal{M}_\pi = G_{\tau_1}, G_{\tau_2}, G_{\tau_3}, \ldots$.*

2.3 Composition of Size-Change Graphs

Definition 6. *The* composition *of two size-change graphs $G : \mathtt{f} \to \mathtt{g}$ and $G' : \mathtt{g} \to \mathtt{h}$ is $G; G' : \mathtt{f} \to \mathtt{h}$ with arc set E defined below. Notation: write $\mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z}$ if $\mathtt{x} \xrightarrow{r} \mathtt{y}$ and $\mathtt{y} \xrightarrow{r'} \mathtt{z}$ are respectively arcs of G and G'.*

$$\begin{aligned} E = \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{z} \mid \exists \mathtt{y}, r \ . \ \mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \xrightarrow{r} \mathtt{z} \ \mathit{or} \ \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{\downarrow} \mathtt{z}\} \\ \bigcup \{\mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z} \mid (\exists \mathtt{y} \ . \ \mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{y} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z}) \ \mathit{and} \ \ (\forall \mathtt{y}, r, r' \ . \ \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z} \ \mathit{implies} \ r = r' = \overset{=}{\downarrow})\} \end{aligned}$$

Further, we define:

- *Size-change graph G is* idempotent *if $G; G = G$.*
- *$G_\pi = G_{\tau_1}; \ldots; G_{\tau_n}$ for any finite call sequence $\pi = \tau_1 \ldots \tau_n \in \mathcal{T}^*$.*

Lemma 7. *The composition operator "$;$" is associative.*

2.4 A Closure Algorithm to Decide the SCT Property

Definition 8. *The* closure *of a set $\mathcal{G}$ of size-change graphs is the smallest set $cl(\mathcal{G})$ such that*

- *$\mathcal{G} \subseteq cl(\mathcal{G})$*
- *If $G_1 : \mathtt{f} \to \mathtt{f}'$ and $G_2 : \mathtt{f}' \to \mathtt{f}''$ are in $cl(\mathcal{G})$, then $G_1; G_2 \in cl(\mathcal{G})$.*

In the worst case, $cl(\mathcal{G})$ can be exponentially larger than $\mathcal{G}$, see [27].

Example 9. Suppose $\mathcal{G} = \{G_1, G_2, G_3\}$ as in Example 1. Its closure is

$$cl(\mathcal{G}) = \{G_1, G_2, G_3, G_{12}, G_{123}, G_{1231}, G_{13}, G_{131}, G_{1312}, G_{23}, G_{231}, G_{31}\}$$

Each graph in $cl(\mathcal{G})$ is the composition G_π for a finite P call sequence π, e.g.,

$$G_{231} = G_2; G_3; G_1$$

for $\pi = 231$ has $\mathtt{f}$ as both source and target, and contains one arc: $\mathtt{u} \xrightarrow{\downarrow} \mathtt{u}$.

Theorem 10. *Program P is SCT terminating iff every idempotent G in $cl(\mathcal{G})$ has an arc $z \xrightarrow{\downarrow} z$.*

Proof. This is Theorem 4 from [27]. For "only if" ($\Rightarrow$), suppose P is size-change terminating and that G_π in $cl(\mathcal{G})$ is idempotent: $G_\pi = G_\pi; G_\pi$. By Definition 5, the infinite call sequence $\pi^\omega = \pi, \ldots, \pi, \pi, \ldots$ has an infinitely descending thread. Consider this thread's position at the start of each π in π^ω. There are finitely many variables, so the thread must visit some variable `x` infinitely often. Thus there must be $n, \mathtt{x}$ such that π^n has a thread from `x` to `x` containing $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$. By Definition 6, arc $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$ is in G_{π^n}. Idempotence of G_π implies $G_{\pi^n} = G_\pi; G_\pi; \ldots; G_\pi = G_\pi$, so $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$ is in G_π.

"If" ($\Leftarrow$): we show that if P is *not* size-change terminating, there exists an idempotent $G \in cl(\mathcal{G})$ without an arc $z \xrightarrow{\downarrow} z$. Assuming P is not size-change terminating, by Definition 5 there is an infinite call sequence $\pi = \tau_1\tau_2\ldots$ such that multipath $\mathcal{M}_\pi = G_{\tau_1}, G_{\tau_2}, \ldots$ has no infinitely descending thread. Define

$$h(k, \ell) = G_{\tau_k}; \ldots; G_{\tau_{\ell-1}}$$

for $k, \ell \in I\!\!N$ with $0 < k < \ell$. Define equivalence relation $\simeq$ on h's domain by

$$(k, \ell) \simeq (k', \ell') \text{ if and only if } \; h(k, \ell) = h(k', \ell')$$

Relation $\simeq$ is of finite index since the closure set $cl(\mathcal{G})$ is finite. By Ramsey's Theorem there exists an infinite set $K \subseteq I\!\!N$ and fixed $m, n \in I\!\!N$ such that $(k, \ell) \simeq (m, n)$ for any $k, \ell \in K$ with $k < \ell$. Expanding the definition of $\simeq$ gives

$$G_{\tau_k}; \ldots; G_{\tau_{\ell-1}} = G_{\tau_m}; \ldots; G_{\tau_{n-1}}$$

Let $G^\circ = h(m, n)$. By associativity of ;, $p, q, r \in K$, with $p < q < r$ implies

$$\begin{aligned} G^\circ &= G_{\tau_p}; \ldots; G_{\tau_{r-1}} \\ &= (G_{\tau_p}; \ldots; G_{\tau_{q-1}}); (G_{\tau_q}; \ldots; G_{\tau_{r-1}}) \\ &= G^\circ; G^\circ \end{aligned}$$

so G° is idempotent (and so $G^\circ : \mathtt{f} \to \mathtt{f}$ for some `f`).

If G° had an arc $z \xrightarrow{\downarrow} z$, then the multipath $G_{\tau_m}, \ldots, G_{\tau_{n-1}}$ would have a descending thread from z to z. This would imply $\mathcal{M}_\pi$ has an infinite descending thread, violating the assumption about π. □

Theorem 11. *The problem of deciding SCT termination is in* PSPACE *(as a function of program size).*

Proof. (Sketch) First, we argue that SCT termination is a path property in a certain graph. Since the composition operator " ; " is associative, a graph G is in $cl(\mathcal{G})$ iff $G = G_\pi$ for some π. Thus by Theorem 10

> P is size-change terminating iff there exists no call sequence π such that G_π is idempotent and G_π contains no arc $z \xrightarrow{\downarrow} z$.

This is a reachability problem in a directed graph (call it Γ). Each node of Γ is a size-change graph G, and each arc is from G_π to $G_{\pi\,\tau}$ where $\pi \in \mathcal{T}^*, \tau \in \mathcal{T}$. The number of nodes in Γ is the number of possible size-change graphs G for program P.

A well-known result by Savitch is that existence of a path in a directed graph with m nodes can be decided[2] in space $O(\log^2 m)$. (See [23] for the "divide-and-conquer" proof.) The number of size-change graphs is bounded by 3^{p^2} where p is the number of variables in P. (Reasoning: between any two variables there may be no arc, or one arc labeled by $\downarrow$, or one labeled by $\overline{\downarrow}$.) Thus the graph may, by Savitch's result, be searched using memory space $O(\log^2(3^{p^2}))$. This is clearly bounded by a polynomial in the number of variables of program P. □

[27] shows PSPACE to be a lower bound, so the problem is PSPACE-complete.

3 Transition Invariants (TI)

3.1 Programs Defined by Transitions

Following [31,28], in order to abstract away from the syntax of imperative programs we use transitions to formalize programs. A transition τ can be thought of as a label or a statement.

Definition 12 (Transition-based program). *We define a* program *to be a triple*

$$P = (\Sigma, \mathcal{T}, \rho),$$

consisting of:

- *a set of* states Σ,
- *a finite set of* transitions $\mathcal{T}$, *and*
- *a function ρ that assigns to each transition a binary relation over states,*

$$\rho_\tau \subseteq \Sigma \times \Sigma, \qquad \textit{for} \;\; \tau \in \mathcal{T}.$$

The transition relation of P, *denoted R_P, comprises the transition relations ρ_τ of all transitions $\tau \in \mathcal{T}$, i.e.,*

$$R_P = \bigcup_{\tau \in \mathcal{T}} \rho_\tau .$$

A program P is *terminating* if its transition relation R_P is well-founded. This means there is no infinite computation

$$s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{\tau_2} s_3 \xrightarrow{\tau_3} \ldots$$

i.e., there is no sequence of states $s_1, s_2, \ldots$ and transitions $\tau_1, \tau_2, \ldots$ such that for every $i \in \mathbb{N}$, the state pair (s_i, s_{i+1}) is contained in the transition relation ρ_{τ_i}.

[2] A key point is that the entire graph Γ is not held in storage at any time, but just the nodes currently being investigated. See [23] for the "divide-and-conquer" proof.

States. Imperative programs in most references use a more concrete version of states:

$$\Sigma = \mathsf{Loc} \times (\mathsf{Var} \to \mathsf{Value})$$

where $\mathsf{Loc}, \mathsf{Var}$ are finite sets (of *locations* and *variables*), and Value is a perhaps infinite set of *values*. Typical elements (perhaps decorated) are: $\ell \in \mathsf{Loc}, z \in \mathsf{Var}$ and $v \in \mathsf{Value}$. A state in Σ has form $s = (\ell, \sigma)$ where $\sigma : \mathsf{Var} \to \mathsf{Value}$.

We only deal with one program at a time, so the objects $P, \mathsf{Loc}, \mathsf{Var}, \mathsf{Value}$ will have fixed values. Letting $\mathsf{Var} = \{z_1, \ldots, z_m\}$, a state can be written as $s = (\ell, \sigma)$ or

$$s = (\ell, v_1, \ldots, v_m)$$

Here ℓ is the current location; $v_1, \ldots, v_m \in \mathsf{Value}$ are the respective values of variables $z_1, \ldots, z_m$.

Extensional versus intensional representation. Program P's semantics is by Definition 12 its transition relation $R_P \subseteq \Sigma \times \Sigma$, that is, a set of pairs of states (s, s'), where s is the "current state" and s' is the "next state". Natural operations on such sets include boolean operations $\cup, \cap, \setminus$ and set-formers. This corresponds to an *extensional* view of semantics.

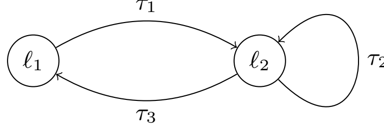
For practical uses (e.g., in a theorem prover) many writers use as alternative an *intensional* view of semantics, and represent a set of state-pairs, i.e., a transition relation, by a logic formula with implicit universal quantification over free logical variables. The universe of discourse (for a given, fixed, program P): a formula will *always* denote a subset of $\Sigma \times \Sigma$. Formulas are built using logical operations $\vee, \wedge, \Rightarrow, \neg$. These correspond exactly to $\cup, \cap, \subseteq$ and $\Sigma \setminus _$.

We follow the usual convention of naming values in s by unprimed logical variables, and values in s' by primed logical variables. Logical variables are program counters pc, pc' and the variables of program P. Locations: the atomic formula $pc = \ell$ means that the control location of s is ℓ; and $pc' = \ell'$ means that the control location of s' is ℓ'. Formula variables other than pc, pc' are program variables ranging over Value. Formulas can represent state pair sets compactly, since variables not occurring in a formula are simply not constrained (they range over all of Value or Loc).

Example 13. Figure 3 expresses a transition-based program in the sense of Definition 12, using an intensional representation, and comma to abbreviate conjunction $\wedge$. As we will see in Section 4, the program stems from translating the functional program from **Example 1**.

3.2 Termination by Transition Invariants

In this section we give a brief description of terminology and results of [31] restricted to termination ([31] also deals with general liveness properties and fairness). We write r^+ to denote the transitive closure of a relation r.



ρ_{τ_1} is $pc = \ell_1, pc' = \ell_2,\ x \neq 0,\ x' = x,\quad y' = y,\ u' = x, v' = y,\quad w' = y$
ρ_{τ_2} is $pc = \ell_2, pc' = \ell_2,\ v > 0,\ x' = x,\quad y' = y,\ u' = u, v' = v - 1, w' = 2 * w$
ρ_{τ_3} is $pc = \ell_2, pc' = \ell_1,\ v = 0,\ x' = u - 1, y' = w, u' = u, v' = v,\quad w' = w$

Fig. 3. Transition relation and corresponding control flow graph of a program $P = (\Sigma, \mathcal{T}, \rho)$ where the set of states Σ is $\{\ell_1, \ell_2\} \times \mathbb{N}^5$, the set of transitions $\mathcal{T}$ is $\{\tau_1, \tau_2, \tau_3\}$ and the program's transition relation $R_P(pc, x, y, u, v, w, pc', x', y', u', v', w')$ is $\rho_{\tau_1} \vee \rho_{\tau_2} \vee \rho_{\tau_3}$

Definition 14 (Transition invariant). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$, a transition invariant T is a binary relation over states T that contains the transitive closure R_P^+ of the program's transition relation R_P, i.e.,*

$$R_P^+ \subseteq T.$$

Definition 15 (Disjunctively well-founded relation). *A relation T is* disjunctively well-founded *if it is a finite union of well-founded relations:*

$$T = T_1 \cup \cdots \cup T_n$$

Theorem 16 (Proof rule for termination). *A program P is terminating if and only if there exists a disjunctively well-founded transition invariant for P.*

As a consequence of the above theorem, we can prove termination of a program P as follows.

1. Find a finite number of relations $T_1, \ldots, T_n$.
2. Show that the inclusion $R_P^+ \subseteq T_1 \cup \cdots \cup T_n$ holds.
3. Show that each relation $T_1, \ldots, T_n$ is well-founded.

Proof. This is Theorem 1 from [31]. "Only if" ($\Rightarrow$) is trivial: if P is terminating, then both R_P and R_P^+ are well-founded. Choose $n = 1$ and $T_1 = R_P^+$.

"If" ($\Leftarrow$): we show that if P is *not* terminating and $T_1 \cup \cdots \cup T_n$ is a transition invariant, then some T_i is not well-founded. Nontermination of P means there exists an infinite computation:

$$s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} s_3 \xrightarrow{\tau_3} \ldots$$

Let choice function f satisfy

$$f(k, \ell) \in \{\ T_i \mid (s_k, s_\ell) \in T_i\ \}$$

for $k, \ell \in \mathbb{N}$ with $k < \ell$. (The condition $R_P^+ \subseteq T_1 \cup \cdots \cup T_n$ implies that f exists, but does not define it uniquely.) Define equivalence relation $\simeq$ on f's domain by

$$(k, \ell) \simeq (k', \ell') \text{ if and only if } f(k, \ell) = f(k', \ell')$$

Relation $\simeq$ is of finite index since the set of T's is finite. By Ramsey's Theorem there exists an infinite sequence of natural numbers $k_1 < k_2 < \ldots$ and fixed $m, n \in I\!N$ such that

$$(k_i, k_{i+1}) \simeq (m, n) \qquad \text{for all } i \in I\!N.$$

Hence $(s_{k_i}, s_{k_{i+1}}) \in T_{mn}$ for all i. This is a contradiction: T_{mn} is not well-founded. □

In comparison to Theorem 10 the proof of Theorem 16 uses a weaker version of Ramsey's theorem. The weak version of Ramsey's theorem states that every infinite complete graph that is colored with finitely many colors contains a monochrome infinite path.

Example 17. Consider the program P in Figure 3 and the binary relations $T_1, \ldots, T_5$ given by the five formulas below.

$$\begin{aligned}
T_1 :&\quad pc = \mathtt{f} \;\wedge\; pc' = \mathtt{f} \;\wedge\; x > x',\\
T_2 :&\quad pc = \mathtt{g} \;\wedge\; pc' = \mathtt{g} \;\wedge\; v > v',\\
T_3 :&\quad pc = \mathtt{g} \;\wedge\; pc' = \mathtt{g} \;\wedge\; u > u',\\
T_4 :&\quad pc = \mathtt{f} \;\wedge\; pc' = \mathtt{g},\\
T_5 :&\quad pc = \mathtt{g} \;\wedge\; pc' = \mathtt{f},
\end{aligned}$$

The union of these relation is a transition invariant for P, i.e., the inclusion $R_P^+ \subseteq T_1 \cup \cdots \cup T_5$ holds. Since every T_i is well-founded, their union is a disjunctively well-founded transition invariant and hence, by Theorem 16 the program P is terminating.

3.3 Transition Predicate Abstraction (TPA)

Transition predicate abstraction [32] is a method to compute transition invariants, just as predicate abstraction is a method to compute invariants. The method takes as input a selection of finitely many binary relation over states. We call these relations *transition predicates*. For this section we fix a finite set of transition predicates $\mathcal{P}$. We usually refer to a transition predicate by the formula that defines it.

Definition 18 (Set of abstract transitions $\mathcal{T}_{\mathcal{P}}^{\#}$). *Given the set of transition predicates $\mathcal{P}$, the set of* abstract transitions *$\mathcal{T}_{\mathcal{P}}^{\#}$ is the set that contains the conjunction of every subset of transition predicates $\{p_1, \ldots p_m\} \subseteq \mathcal{P}$, i.e.,*

$$\mathcal{T}_{\mathcal{P}}^{\#} = \{p_1 \wedge \ldots \wedge p_m \mid p_i \in \mathcal{P},\ 0 \le m, 1 \le i \le m\}$$

Clearly $\mathcal{T}_{\mathcal{P}}^{\#}$ is closed under intersection, and the set of all binary relations over states $\Sigma \times \Sigma$ is a member of $\mathcal{T}_{\mathcal{P}}^{\#}$ (the case $m = 0$).

Example 19. Consider the following set of transition predicates.

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

The set of abstract transitions $\mathcal{T}^{\#}_{\mathcal{P}}$ is

$$\{\mathsf{true},\quad x = x',\quad x > x',\quad y > y',\quad x = x' \wedge y > y',\quad x > x' \wedge y > y',\ \mathsf{false}\}$$

The abstract transition written as true is the set of all state pairs $\Sigma \times \Sigma$ and is the empty conjunction of transition predicates. The abstract transition false is the empty relation; e.g., the conjunction of $x = x'$ and $x > x'$ is false.

We next define a function that assigns to a binary relation T over states the least (wrt. inclusion) abstract transition that is a superset of T.

Definition 20 (Abstraction function α). *A set of transition predicates $\mathcal{P}$ defines the* abstraction function

$$\alpha : 2^{\Sigma \times \Sigma} \rightarrow \mathcal{T}^{\#}_{\mathcal{P}}$$

that assigns to a relation $r \subseteq \Sigma \times \Sigma$ the smallest abstract transition that is a superset of r, i.e.,

$$\alpha(r) = \bigwedge\{p \in \mathcal{P} \mid r \subseteq p\}.$$

We note that α is extensive, i.e., the inclusion

$$r \subseteq \alpha(r)$$

holds for any binary relation over states $r \subseteq \Sigma \times \Sigma$.

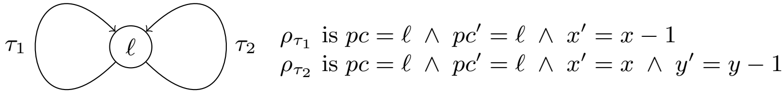


Fig. 4. Transition relation and corresponding control flow graph of a program $P = (\Sigma, \mathcal{T}, \rho)$ where the set of states Σ is $\{\ell\} \times \mathbb{N} \times \mathbb{N}$, the set of transitions $\mathcal{T}$ is $\{\tau_1, \tau_2\}$ and the program's transition relation $R_P(pc, x, y, pc', x', y')$ is $\rho_{\tau_1} \vee \rho_{\tau_2}$

Example 21. Application of the abstraction function α to the transition relations ρ_1 and ρ_2 of the program in Figure 4 results in the following abstract transitions.

$$\begin{array}{lll} \alpha(\rho_{\tau_1}) & \text{is} & x > x' \\ \alpha(\rho_{\tau_2}) & \text{is} & x = x' \wedge y > y' \end{array}$$

We next present an algorithm that uses the abstraction α to compute (a set of abstract transitions that represents) a transition invariant. The algorithm terminates because the set of abstract transitions $\mathcal{T}^{\#}_{\mathcal{P}}$ is finite.

Algorithm (TPA)
Transition invariants via transition predicate abstraction.

Input: program $P = (\Sigma, \mathcal{T}, \rho)$
set of transition predicates $\mathcal{P}$
abstraction α defined by $\mathcal{P}$ (according to Def. 20)

Output: set of abstract transitions $P^\# = \{T_1, \ldots, T_n\}$
such that $T_1 \cup \cdots \cup T_n$ is a transition invariant

$P^\# := \{\alpha(\rho_\tau) \mid \tau \in \mathcal{T}\}$
repeat
$P^\# := P^\# \cup \{\alpha(T \circ \rho_\tau) \mid T \in P^\#,\ \tau \in \mathcal{T},\ T \circ \rho_\tau \neq \emptyset\}$
until no change

Our notation $P^\#$ for the set of abstract transitions computed by Algorithm TPA stems from [32]. There, $P^\#$ is called an abstract transition program. In contrast to [32] we do not consider edges between the abstract transitions.

Theorem 22 (TPA). *Let $P^\# = \{T_1, \ldots, T_n\}$ be the set of abstract transitions computed by Algorithm* TPA*. If every abstract relation $T_1, \ldots, T_n$ is well-founded, then program P is terminating.*

Proof. The union of the abstract relations $T_1 \cup \cdots \cup T_n$ is a transition invariant. If every abstract relation $T_1, \ldots, T_n$ is well-founded, the union $T_1 \cup \cdots \cup T_n$ is a disjunctively well-founded transition invariant and by Theorem 16 the program P is terminating. □

Example 23. Consider the program P in Figure 4 and the set of transition predicates $\mathcal{P}$ in Example 19. The output of Algorithm TPA is

$$P^\# = \{x > x', \quad x = x' \wedge y > y'\}$$

Both abstract transitions in $P^\#$ are well-founded. Hence P is terminating.

4 Correctness Proofs and Abstractions

We have defined size-change termination for functional programs, and transition invariants for transition-based programs. For the purpose of comparison, we now restrict size-change termination to the transition-based programs that we obtain from translating functional programs.

From now on, we deal only with *tail-recursive* functional programs where all functions use a common variable name space (the latter condition is not a proper restriction since we can always add redundant parameters, and rename parameters if necessary to ensure uniqueness).

Under this restriction, the translation of a functional program into a transition-based program $P = (\Sigma, \mathcal{T}, \rho)$ with the same termination behavior is straightforward:

- the set of states Σ is the Cartesian product of the set of locations Loc and the data domains for the function parameters; we have a location $\ell_{\mathtt{f}}$ in Loc for every function `f`,
- the set of transitions $\mathcal{T}$ contains a transition τ_c for each call c,
- the transition relation ρ_{τ_c} of the transition τ_c is defined by

$$\rho_{\tau_c} = \{((\ell_{\mathtt{f}}, \boldsymbol{v}), (\ell_{\mathtt{g}}, \boldsymbol{w})) \mid \mathcal{E}[\![\mathtt{e}_1]\!]\boldsymbol{v} = w_1, \ldots, \mathcal{E}[\![\mathtt{e}_n]\!]\boldsymbol{v} = w_n\}.$$

  if the call c occurs in a function definition of the form

$$\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \ldots c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \ldots \ .$$

From now on we fix a transition-based program P which stems from the translation of a (tail-recursive) functional program. The size-change termination of P is equivalent to the size-change termination of the original functional program.

Example 24. The translation-based program in Figure 3 is obtained by translating the functional program in Figure 1 after adding parameters in order to obtain a common variable name space.

```
f(x,y,u,v,w) = if x=0 then y else 1: g(x,y,x,y,y)
g(x,y,u,v,w) = if v>0 then 2: g(x,y,u,v-1,2*w)
                          else 3: f(u-1,w,u,v,w)
```

Fig. 5. The functional program of Figure 1, modified to have a common name space

4.1 From Graphs to Transition Relations

Suppose size-change graph G safely describes call c as in Definition 3. Clearly G expresses a conjunction of relations (each one either $\downarrow$ or $\overset{=}{\downarrow}$) between some parameters of source `f` and some parameters of target `g`. By the common name space assumption, `f` and `g` have the same parameters. This section shows that graph G defines an abstraction of c's transition relation ρ_{τ_c}.

Since a graph is not a set of pairs of states (and not a formula either), we devise a notation $\Phi(G)$ for the set of state pairs described by size-change graph G. Therefore we define as a first step a class of binary relations that represent the atomic pieces of information contained in a size-change graph (which are: source, target, value decrease and strict value decrease).

Definition 25 (Set of size-change predicates $\mathcal{P}_{SCT}$). *We call a binary relation a* size-change predicate *if it is defined by one of the formulas*

$$pc = \ell$$
$$pc' = \ell$$
$$z_i \geq z'_j$$
$$z_i > z'_j$$

where the variable pc ranges over the set of program locations Loc*, and* z_i *and* z_j *are program variables. We use* $\mathcal{P}_{SCT}$ *for the (finite) set of all size-change predicates for the current program* P*.*

As a second step, we define the relation $\Phi(G)$ to be a conjunction of these size-change predicates (parallel to Definition 2).

Definition 26. *Given a size-change graph* $G : \ell \to \ell'$ *with arc set* E*, define the binary relation over states* $\Phi(G) \subseteq \Sigma \times \Sigma$ *by the following formula.*

$$pc = \ell \wedge pc' = \ell' \wedge \bigwedge \{z_i \geq z_j' \mid (z_i, \overline{\downarrow}, z_j) \in E\} \wedge \bigwedge \{z_i > z_j' \mid (z_i, \downarrow, z_j) \in E\}$$

Example 27. The binary relations over states assigned to the size-change graphs of Figure 5 are the following.

$\Phi(G_1)$ is $pc = \mathtt{f} \wedge pc' = \mathtt{g} \wedge\ x \geq x' \wedge\ y \geq y' \wedge\ \ x \geq u' \wedge\ y \geq v' \wedge\ y \geq w'$
$\Phi(G_2)$ is $pc = \mathtt{g} \wedge pc' = \mathtt{g} \wedge\ x \geq x' \wedge\ y \geq y' \wedge\ \ u \geq u' \wedge\ v > v'$
$\Phi(G_3)$ is $pc = \mathtt{g} \wedge pc' = \mathtt{f} \wedge\ u > x' \wedge\ w \geq y' \wedge\ u \geq u' \wedge\ v \geq v' \wedge\ w \geq w'$

The inclusion $R_P \subseteq \Phi(G_1) \vee \Phi(G_2) \vee \Phi(G_3)$ means that the transition relation R_P is approximated by the set $\{G_1, G_2, G_3\}$ of size-change graphs. The inclusion is strict, meaning that the approximation loses precision. An instance of precision loss: the set $\mathcal{P}_{SCT}$ does not contain any of the transition predicates $x \neq 0$, $v > 0$, $v = 0$ that account for the tests.

The following definition extends Definition 3 from a functional program to its translation to a transition-based program P.

Definition 28. *Let* G_τ *be the size-change graph assigned to the transition* τ *of program* P*. We say that* G_τ *is safe for* τ *if the inclusion* $\rho_\tau \subseteq \Phi(G_\tau)$ *holds. A set of graphs* $\{G_\tau \mid \tau \in \mathcal{T}\}$ *is a* safe description *of program* P *if* G_τ *is safe for* τ *for every transition* τ *of* P*.*

We now consider the composition of size-change graphs (Definition 6).

Lemma 29. *The composition of the two size-change graphs* $G_1 : \ell \to \ell'$ *and* $G_2 : \ell' \to \ell''$ *overapproximates the composition of the relations they define, i.e.,*

$$\Phi(G_1) \circ \Phi(G_2) \subseteq \Phi(G_1; G_2).$$

Corollary 30. *If* G_τ *is a size-change graph that is safe for* τ*, then for every transition relation* T *and every size-change graph* G *such that* $G; G_\tau$ *is defined*

$$T \subseteq \Phi(G) \textit{ implies } T \circ \rho_\tau \subseteq \Phi(G; G_\tau)$$

Proof. $\rho_\tau \subseteq \Phi(G_\tau)$ by Definition 28, so $T \circ \rho_\tau \subseteq \Phi(G) \circ \Phi(G_\tau) \subseteq \Phi(G; G_\tau)$ by Lemma 29. □

Lemma 31. *Let* G *be a size-change graph such that source and target of* G *coincide. If* G *has an arc of form* $x \xrightarrow{\downarrow} x$ *then the relation* $\Phi(G)$ *is well-founded.*

Proof. Let G be a size change graph with an an arc $x \xrightarrow{\downarrow} x$. By Definition 26 the relation $\Phi(G)$ is a subset of the relation $x' < x$. Since $x' < x$ is well-founded, all subsets are also well-founded. □

Remark: The converse of Lemma 31 is false, e.g., for the arc set $\{x \xrightarrow{\downarrow} y, y \xrightarrow{\downarrow} x\}$.

4.2 SCT and Disjunctive Well-Foundedness

Theorem 32 (Idempotence and well-foundedness). *If every* idempotent *size-change graph in the closure $cl(\mathcal{G})$ of the set of size-change graphs $\mathcal{G}$ defines a well-founded relation, i.e.,*

$$\forall G \in cl(\mathcal{G}) : \; G; G = G \implies \Phi(G) \textit{ well-founded}$$

then $\Phi(G)$ is well-founded for every *graph in $cl(\mathcal{G})$.*

Proof. Let $G \in cl(\mathcal{G})$ be a size-change graph.

Case 1: Source and target of G do not coincide.
Then there exist two different locations ℓ, ℓ' such that $\Phi(G) \subseteq pc = \ell \wedge pc' = \ell'$ and therefore $\Phi(G) \circ \Phi(G) = \emptyset$ which implies that $\Phi(G)$ is well-founded.
Case 2: Source and target of G coincide.
Then G^n is defined for all $n \in I\!N$. The semigroup $(\{G^n \mid n \in \mathbb{N}\}, ;)$ is finite and has therefore an idempotent element G^k (since every finite semigroup has an idempotent element). By assumption $\Phi(G^k)$ is well-founded. By induction over k and Lemma 29 the inclusion $\Phi(G)^k \subseteq \Phi(G^k)$ holds. Hence $\Phi(G)^k$ is well-founded and therefore also $\Phi(G)$ is well-founded (Reason: If a relation r is not well-founded, then for all $n \in \mathbb{N}$, r^n is not well-founded.)

Therefore, for every $G \in cl(\mathcal{G})$ the transition $\Phi(G)$ is well-founded. □

Since size-change termination is equivalent to the premise of Theorem 32, and its conclusion can be expressed in terms of disjunctive well-foundedness, we obtain the following statement directly.

Corollary 33 (SCT and disjunctive well-foundedness). *Let $\mathcal{G}$ be a set of size-change graphs that is a safe description of program P. If program P is size-change terminating for a set of size-change graphs $\mathcal{G}$ that is a safe description of P, then the relation defined by its closure $cl(\mathcal{G})$*

$$\bigcup \{\Phi(G) \mid G \in cl(\mathcal{G})\}$$

is a disjunctively well-founded transition invariant for P.

Proof. We first show, that the disjunction is a transition invariant, i.e.,

$$R_P^+ \subseteq \bigcup \{\Phi(G) \mid G \in cl(\mathcal{G})\}.$$

Let $(s, s') \in R_P^+$. By definition of R_P^+ there is a sequence of transition relations $\rho_{\tau_1}, \rho_{\tau_2}, \ldots, \rho_{\tau_n}$ such that (s, s') is contained in the composition $\rho_{\tau_1} \circ \rho_{\tau_2} \circ \cdots \circ \rho_{\tau_n}$.

For every such sequence there is a size-change graph $G \in cl(\mathcal{G})$ such that the inclusion $\rho_{\tau_1} \circ \rho_{\tau_2} \circ \cdots \circ \rho_{\tau_n} \subseteq \Phi(G)$ holds. This can be shown by induction

over n, where the induction basis holds by Definition 28 and the induction step follows from Corollary 30. Hence $(s, s') \in \Phi(G)$ for some $G \in cl(\mathcal{G})$, so $(s, s') \in \bigcup\{\Phi(G) \mid G \in cl(\mathcal{G})\}$.

Since P is size-change terminating for $\mathcal{G}$, every idempotent size change-graph $G \in cl(\mathcal{G})$ contains an arc of form $x \xrightarrow{\downarrow} x$ (by Theorem 10, or Theorem 4 of [27]). By Lemma 31, for every idempotent size change-graph $G \in cl(\mathcal{G})$ the relation $\Phi(G^k)$ is well-founded. Hence by Theorem 32 for every size change-graph $G \in cl(\mathcal{G})$ the relation $\Phi(G^k)$ is well-founded. Therefore $\bigcup \{\Phi(G) \mid G \in cl(\mathcal{G})\}$ is a disjunctively well-founded transition invariant for P. □

4.3 Size-Change Graphs and Transition Predicate Abstraction

Lemma 34. *Let α be the abstraction function for the set of size-change predicates $\mathcal{P}_{SCT}$. If the size-change graph G denotes a superset of a binary relation over states T, then the size-change graph G denotes a superset of the abstract transition $\alpha(T)$, i.e.*

$$T \subseteq \Phi(G) \quad \textit{implies} \quad \alpha(T) \subseteq \Phi(G)$$

Proof. For every size-change graph G, the relation $\Phi(G)$ is a conjunction of size-change predicates. Therefore the inclusion $\alpha(T) \subseteq \Phi(G)$ holds if for every $p \in \mathcal{P}_{SCT}$ the inclusion $\Phi(G) \subseteq p$ implies the inclusion $\alpha(T) \subseteq p$. Let p be a size-change predicate. Let $\Phi(G) \subseteq p$. Assume that the inclusion $T \subseteq \Phi(G)$ holds. Then the inclusion $T \subseteq p$ holds and by Definition 20 the inclusion $\alpha(T) \subseteq p$ holds.

Corollary 35. *Let α be the abstraction function for the set of size-change predicates $\mathcal{P}_{SCT}$. The abstract transition $\alpha(\rho_\tau)$ is a subset of the denotation of any size-change graph G_τ that is safe for τ, formally*

$$\alpha(\rho_\tau) \subseteq \Phi(G_\tau)$$

This inclusion can be strict in case G_τ is not the "best" description of ρ_τ. An extreme example: G_τ has the empty set of arcs.

Lemma 36. *Let $cl(\mathcal{G})$ be the closure (Definition 8) for a set of size-change graphs $\mathcal{G}$ that is a safe description of program P. Let $P^{\#}$ be a set of abstract transitions computed by Algorithm* TPA *for the set of size-change predicates $\mathcal{P}_{SCT}$.*

For every abstract transition T in $P^{\#}$ there exists a size-change graph G in $cl(\mathcal{G})$ that contains T, formally

$$\Phi(G) \supseteq T.$$

Proof. Let $T \in P^{\#}$. By Algorithm TPA there is a sequence of transitions $\tau_1, \ldots, \tau_2$ such that the equation

$$T = \alpha(\ldots \alpha(\alpha(\rho_{\tau_1}) \circ \rho_{\tau_2}) \cdots \circ \rho_{\tau_n})$$

holds. Let G_{τ_i} be a graph that is safe for τ_i and G be a size-change graph defined by the following equation.

$$G = G_{\tau_1}; \ldots ; G_{\tau_n}$$

The inclusion $\Phi(G) \supseteq T$ holds by induction, where the induction basis holds by Definition 28 and Lemma 34 and the induction step follows from Corollary 30.

Theorem 37. *Let $\mathcal{G}$ be a set of size-change graphs that is a safe description of program P. Let $P^{\#}$ be a set of abstract transitions computed by Algorithm* TPA *for the set of size-change predicates $\mathcal{P}_{SCT}$. If P is size-change terminating for $\mathcal{G}$ then $P^{\#}$ defines a disjunctively well-founded transition invariant.*

Proof. The output of Algorithm TPA $P^{\#}$ defines a transition invariant for P. If P is size-change terminating, then by Corollary 33 every element of $\{\Phi(G) \mid G \in cl(\mathcal{G})\}$ is well-founded. Hence by Lemma 36 every element of $P^{\#}$ is well-founded. Therefore $\bigcup P^{\#}$ is a disjunctively well-founded transition invariant. □

5 Decision Problems for Termination Analyses

In this section, we categorize the base algorithm in the different termination analyses by the decision problem that it solves, and then establish an formal connection between the decision problems.

Part of the input of those decision problems will be a *transition abstraction*. A transition abstraction fixes a set of abstract values $\mathcal{T}^{\#}$ and their meaning via the *denotation* function γ. Each abstract value a denotes a relation over states, i.e., $\gamma(a) \subseteq \Sigma \times \Sigma$. The transition abstraction fixes also a distinguished abstract value a_τ for every transition τ of the given program. A termination analysis starts with those values.

Programs, transition relations, states, etc. are as defined in Section 3.

Definition 38 (Transition Abstraction). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$, a* transition abstraction *is a triple*

$$(\mathcal{T}^{\#}, \gamma, \{a_\tau \,|\, \tau \in \mathcal{T}\})$$

consisting of:

1. *a finite set $\mathcal{T}^{\#}$ of abstract values called* abstract relations*,*
2. *a* denotation *function γ that assigns to each abstract relation a relation over the program's states, i.e.,*

$$a \in \mathcal{T}^{\#} \implies \gamma(a) \subseteq \Sigma \times \Sigma$$

3. *a set of distinguished abstract values indexed by transitions τ of the program, i.e.,*

$$a_\tau \in \mathcal{T}^{\#}, \quad \textit{for} \;\; \tau \in \mathcal{T}.$$

The abstract relation for the transition τ *must safely abstract the transition relation defined by* τ*, formally*

$$\rho_\tau \subseteq \gamma(a_\tau)$$

for each transition τ *in* $\mathcal{T}$*.*

A set X of abstract relations denotes their union, i.e.,

$$\gamma(X) = \bigcup\{\gamma(a) \mid a \in X\}, \quad \text{for} \quad X \subseteq \mathcal{T}.$$

Example 39 (SCT). In order to rephrase *size-change analysis* as presented in Section 2, one may use the transition abstraction where:

- the abstract relations $a \in \mathcal{T}^\#$ are size-change graphs G,
- the denotation function γ is the function Φ of Definition 26, i.e., a graph G denotes the transition relation defined by the formula $\Phi(G)$,
- the distinguished abstract transitions a_τ for transitions τ are exactly the size-change graphs G_τ for calls τ in the set $\mathcal{G}$ fixed in Definition 5.

Since we translate the function Φ on size-change graphs to the denotation function γ, the safety required for the size-changes graphs G_τ translates directly to the safety requirement for the a_τ in Definition 38; see Definition 28.

Example 40 (TPA). In order to rephrase *transition predicate abstraction* as presented in Section 3.3, one may use the transition abstraction where:

- the abstract relations $a \in \mathcal{T}^\#$ are the *abstract transitions* $p_1 \wedge \ldots \wedge p_m$, which are conjunctions of transition predicates $p_j \in \mathcal{P}$, for the given set of transition predicates $\mathcal{P}$.

$$\mathcal{T}^\# = \{p_1 \wedge \ldots \wedge p_m \mid p_1, \ldots, p_m \in \mathcal{P},\ 0 \leq m\}$$

- the denotation function γ is essentially the identity function, i.e., the denotation of a conjunction of transition predicates is the intersection of the transition relations they denote,
- the abstract transition a_τ is the abstraction α applied to the transition relation ρ_τ. This is the strongest abstraction transition that contains ρ_τ, or, equivalently, is the conjunction of all transition predicates in $\mathcal{P}$ that contain ρ_τ; see Definition 20.

$$a_\tau = \alpha(\rho_\tau)\ (= \bigwedge\{p \in \mathcal{P} \mid \rho_\tau \subseteq p\})$$

5.1 Transformer on Abstract Relations

Given a transition τ of the program, we consider a function $F^\#_\tau$ that assigns to each abstract relation a another abstract relation $a' = F^\#_\tau(a)$. The idea is that the function $F^\#_\tau$ abstracts the relational composition with the transition relation ρ_τ (i.e., it abstracts the function F_τ such that $F_\tau(T) = T \circ \rho_\tau$).

For better legibility, we write $F^{\#}(a, \tau)$ instead of $F^{\#}_{\tau}(a)$. We call $F^{\#}$ a (parametrized) abstract-relation transformer.

In this section, we fix a program $P = (\Sigma, \mathcal{T}, \rho)$ and a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \mid \tau \in \mathcal{T}\})$ defining a set of abstract relations, their denotation, and a set of abstract relations for the transitions of the program.

Definition 41 (Abstract-relation transformer $F^{\#}$). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$ and a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \mid \tau \in \mathcal{T}\})$, an* abstract-relation transformer *is a function*

$$F^{\#} : \mathcal{T}^{\#} \times \mathcal{T} \to \mathcal{T}^{\#}$$

such that

$$\gamma(F^{\#}(a, \tau)) \supseteq \gamma(a) \circ \rho_\tau$$

In words, the application of $F^{\#}$ to the abstract relation a and the transition τ overapproximates the relational composition of the relation denoted by a with the transition relation defined by τ.

Example 42 (Continuing Examples 39 and 40)
Continuing Example 39, where we use size-change graphs as abstract relations, one may define the abstract-relation transformer as follows.

$$F^{\#}(G, \tau) = G; G_\tau$$

The safety of G_τ for the call/transition τ yields the safety requirement in Definition 41; see Definition 28, Lemma 29 and Corollary 30.

Continuing Example 40, where we use abstract transitions (i.e., conjunctions of transition predicates) as abstract relations, one may define the abstract-relation transformer by

$$F^{\#}(T, \tau) = \alpha(T \circ \rho_\tau),$$

where α is the abstraction function defined by the given set of transition predicates; see Definition 20.

We next introduce an expression to denote a set of abstract relations. We call it "the least fixpoint of the abstract-relation transformer $F^{\#}$" although, strictly speaking, it is not a fixpoint of the function $F^{\#}$. (Instead, it is the fixpoint of a functional that can be derived from $F^{\#}$. This functional ranges over the powerset lattice generated by the abstract relations. The least fixpoint is the *least* fixpoint of this functional *above* the set $\{a_\tau \mid \tau \in \mathcal{T}\}$, i.e., the set of abstract relations a_τ for the transitions of the given program P. For notational economy we will not formally define the lattice and the functional.)

Definition 43 (Least fixpoint of the abstract-relation transformer $F^{\#}$). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$, a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \mid \tau \in \mathcal{T}\})$, and an abstract-relation transformer $F^{\#}$, the* "least fixpoint of the abstract-relation transformer $F^{\#}$"*, written*

$$\mathit{lfp}(\{a_\tau \mid \tau \in \mathcal{T}\}, F^{\#}),$$

is defined as the least set of abstract relations X such that

– *X contains the set of abstract relations a_τ for each transition τ,*

$$X \supseteq \{a_\tau \,|\, \tau \in \mathcal{T}\}$$

– *and X is closed under application of the abstract-relation transformer for every transition τ, i.e., the application of $F^\#$ to an abstract relation a in X and a transition τ is again an element of X, formally*

$$X \supseteq \{F^\#(a, \tau) \mid a \in X,\ \tau \in \mathcal{T}\}.$$

Lemma 44. *The least fixpoint of the abstract-relation transformer $F^\#$ can be indexed by the sequences of transitions $\tau_1, \ldots, \tau_n$, i.e.,*

$$lfp(\{a_\tau \,|\, \tau \in \mathcal{T}\}, F^\#) = \{a_{\tau_1 \ldots \tau_n} \mid n \geq 1, \tau_1, \ldots, \tau_n \in \mathcal{T}\}$$

where $a_{\tau_1 \tau_2} = F^\#(a_{\tau_1}, \tau_2)$, $a_{\tau_1 \tau_2 \tau_3} = F^\#(a_{\tau_1 \tau_2}, \tau_3)$, etc..

The following lemma states that we can use a transition abstraction and an abstract-relation transformer to compute a transition invariant for the program P.

Lemma 45 (Transition invariants via the abstract-relation transformer $F^\#$). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$ with transition relation R_P, a transition abstraction $(\mathcal{T}^\#, \gamma, \{a_\tau \,|\, \tau \in \mathcal{T}\})$, and an abstract-relation transformer $F^\#$, the least fixpoint of the abstract-relation transformer $F^\#$ denotes a transition invariant for P, i.e.,*

$$R_P^+ \subseteq \gamma(lfp(\{a_\tau \,|\, \tau \in \mathcal{T}\}, F^\#)).$$

We next define a decision problem, and then characterize a specific class of termination analyses as decision procedures for the problem.

Problem: Lfp Checking for Abstract Relations

Input:

– a program $P = (\Sigma, \mathcal{T}, \rho)$
– a transition abstraction $(\mathcal{T}^\#, \gamma, \{a_\tau \,|\, \tau \in \mathcal{T}\})$
– an abstract-relation transformer $F^\# : \mathcal{T}^\# \times \mathcal{T} \to \mathcal{T}^\#$
– a subset $\mathsf{GOOD} \subseteq \mathcal{T}^\#$ such that every element of GOOD denotes a well-founded relation

Property: $lfp(\{a_\tau \,|\, \tau \in \mathcal{T}\}, F^\#) \subseteq \mathsf{GOOD}$

Theorem 46 (Lfp Checking for Abstract Relations and Termination). *The program P is terminating if the decision procedure* Lfp Checking for Abstract Relations *answers yes.*

This decision procedure is a *semi-test* for termination: a yes-answer is definite, a no-answer is no.

Proof. If the procedure answers yes, the least fixpoint of the abstract-relation transformer $F^\#$ is not only a transition invariant (by Lemma 45) but it is also disjunctively well-founded. Thus, Theorem 16 applies and P is terminating. □

Next, a complexity result. To make the statement simpler, we (reasonably) assume henceforth that the number of abstract relations is greater than the number of transitions, i.e., $|\mathcal{T}^\#| \geq |\mathcal{T}|$. In the setting of Examples 39, 40, and 42, the number of abstract relations is:

– (in the setting of SCT, as in Examples 39 and 42) exponential in the square of the size of the program (to be precise, it is bound by 3^{p^2} where p is the number of program variables),
– (in the setting of TPA, as in Examples 40 and 42) exponential in the number of transition predicates in $\mathcal{P}$.

Theorem 47. Lfp Checking for Abstract Relations *is decidable in time polynomial in* $|\mathcal{T}^\#|$*, and (by another algorithm) in space* $\mathcal{O}(log^2|\mathcal{T}^\#|)$*.*

Proof. Consider a directed graph Γ. The nodes of Γ are the abstract relations in $\mathcal{T}^\#$ plus two special nodes *init* and *fin*so the graph Γ contains $|\mathcal{T}^\#| + 2$ nodes. Let $a, a' \in \mathcal{T}^\#$, we define that

– Γ contains an edge from a to a' if and only if there is a $\tau \in \mathcal{T}$ such that $F^\#(a, \tau) = a'$,
– Γ contains an edge from *init* to a if and only if $a \in \{a_\tau \mid \tau \in \mathcal{T}\}$,
– and Γ contains an edge from a to *fin* if and only if $a \notin \mathsf{GOOD}$.

We conclude: Γ contains a path from *init* to a iff $a \in lfp(\{a_\tau \,|\, \tau \in \mathcal{T}\}, F^\#)$. Further, Γ contains a path from *init* to *fin* iff $lfp(\{a_\tau \,|\, \tau \in \mathcal{T}\}, F^\#) \nsubseteq \mathsf{GOOD}$. For time: the graph can be searched by, for example, Dijkstra's algorithm. For space: a well-known result by Savitch is that existence of a path in a directed graph with n nodes can be decided in space $O(\log^2 n)$. □

5.2 Composition of Abstract Relations

In Section 5.1, we used the function $F^\#$ to abstract the relational composition of relations with the transition relations ρ_τ for the program transitions τ. In this section, we introduce a binary operator on abstract relations in order to abstract the binary relational composition operator.

Definition 48 (Abstract composition $\circ^\#$). *Given a program* $P = (\Sigma, \mathcal{T}, \rho)$ *and a transition abstraction* $(\mathcal{T}^\#, \gamma, \{a_\tau \,|\, \tau \in \mathcal{T}\})$*, an* abstract composition *is a binary operation on abstract relations,*

$$\circ^\# : \mathcal{T}^\# \times \mathcal{T}^\# \to \mathcal{T}^\#$$

such that

$$\gamma(a_1 \circ^{\#} a_2) \supseteq \gamma(a_1) \circ \gamma(a_2).$$

In words, the abstract composition of two abstract relations a_1 and a_2 overapproximates the relational composition of the two relations denoted by a_1 resp. a_2.

Example 49 (continuing Examples 39 and 40). In the setting of size-change termination, the composition operator on size-change graphs (written $G_1; G_2$) is an abstract composition by Lemma 29 (it uses Φ for the denotation function γ).

In the setting of transition predicate abstraction, we can define the abstract composition over abstract transitions T_1 and T_2 (i.e., conjunctions of transition predicates) by $T_1 \circ^{\#} T_2 = \alpha(T_1 \circ T_2)$, where α is the abstraction function defined by the given set of transition predicates; see Definition 20. Note that, in contrast with the size-change setting, the abstract composition over abstract transitions is in general not associative.

Definition 50 (Closure of an abstract composition). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$, a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \mid \tau \in \mathcal{T}\})$, and an abstract composition $\circ^{\#}$, the* "closure of the abstract composition $\circ^{\#}$", *written*

$$cl(\{a_\tau \mid \tau \in \mathcal{T}\}, \circ^{\#})$$

is the smallest set of abstract relations X such that

- *X contains the set of abstract relations a_τ for the transitions τ,*

$$X \supseteq \{a_\tau \mid \tau \in \mathcal{T}\}$$

- *and X is closed under abstract composition, i.e., the abstract composition of two abstract relations a_1 and a_2 in X is again an element in X:*

$$X \supseteq \{a_1 \circ^{\#} a_2 \mid a_1 \in X,\ a_2 \in X\}.$$

The following lemma states (in analogy with Lemma 45) that we can use a transition abstraction and an abstract composition over abstract relations to compute a transition invariant for the program P.

Lemma 51 (Transition invariants via abstract composition $\circ^{\#}$). *Given a program $P = (\Sigma, \mathcal{T}, \rho)$ with transition relation R_P, a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \mid \tau \in \mathcal{T}\})$, and an abstract composition $\circ^{\#}$, the closure of the abstract composition denotes a transition invariant for P, i.e.,*

$$R_P^+ \subseteq \gamma(cl(\{a_\tau \mid \tau \in \mathcal{T}\}, \circ^{\#})).$$

In analogy to Section 5.1, we state a decision problem. In contrast with Section 5.1, the input contains not an abstract-relation transformer $F^{\#}$, but an abstract composition $\circ^{\#}$. It is checked if the closure of $\circ^{\#}$ is a subset of GOOD.

This enables us to characterize a second class of termination analyses as decision procedures for this problem.

Problem: Closure Checking for Abstract Relations

Input:

- a program $P = (\Sigma, \mathcal{T}, \rho)$
- a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \,|\, \tau \in \mathcal{T}\})$
- an abstract composition $\circ^{\#} : \mathcal{T}^{\#} \times \mathcal{T}^{\#} \to \mathcal{T}^{\#}$
- a subset $\mathsf{GOOD} \subseteq \mathcal{T}^{\#}$ such that every element of GOOD denotes a well-founded relation

Property: $cl(\{a_\tau \mid \tau \in \mathcal{T}\}) \subseteq \mathsf{GOOD}$

Theorem 52 (Closure Checking for Abstract Relations and Termination). *Program P terminating if the decision procedure* Closure Checking for Abstract Relations *answers yes.*

Proof. Analogous to Theorem 46. □

The next result investigates the complexity of the decision problem Closure Checking for Abstract Relations .

Theorem 53. *The problem* Closure Checking for Abstract Relations *is* ptime*-complete in the number of transition relations $|\mathcal{T}^{\#}|$.*

Proof. First, the problem Closure Checking for Abstract Relations is in ptime, since a straightforward bottom-up algorithm can compute and test for well-foundedness all elements in $cl(\{a_\tau \mid \tau \in \mathcal{T}\})$. (Remark: we count the well-foundedness test $a \in \mathsf{GOOD}$? as one step.)

Second, we show the problem is ptime-hard by reduction from a known ptime-complete problem to Closure Checking for Abstract Relations. The problem gen is a membership problem for the closure of an operation, defined as follows. **Given:** A finite set W, a binary operation *op* on W, a subset $V \subseteq W$, and $w \in W$. **To decide:** Is $w \in cl(V, op)$?

Given a gen instance (W, op, V, w), let $P = (\Sigma, \mathcal{T}, \rho)$ be a program such that

- the set of states is the empty set
- the set of transitions $\mathcal{T}$ is V
- the transition relation ρ_τ of every transition τ is the empty set.

Let $(\mathcal{T}^{\#}, \gamma, \{a_\tau \,|\, \tau \in \mathcal{T}\})$ be a transition abstraction such that

- the set of abstract relations is $\mathcal{T}^{\#} = W$
- the denotation γ assigns to each abstract relation the empty set.
- the set of program transition relations is $\{a_\tau \,|\, \tau \in \mathcal{T}\} = V$.

Since every abstract relation denotes the empty relation, $op = \circ^{\#}$ is trivially an abstract composition. We choose $\mathsf{GOOD} = W \backslash \{w\}$. This is a valid choice since every abstract relation denotes a well-founded relation.

Clearly $w \notin cl(V, op)$ if and only if the inclusion $cl(\{a_\tau \mid \tau \in \mathcal{T}\}, op) \subseteq \mathsf{GOOD}$ holds. The complexity result follows since the negation of any PTIME-complete problem is also PTIME-complete. □

Abstract-relation transformers $F^{\#}$ versus abstract composition $\circ^{\#}$: Precision. A termination analysis A has *higher precision* than a termination analysis B if A returns a yes-answer whenever B does, and possibly strictly more often (a yes-answer is definite in proving termination of the input program).

One might expect, by the complexity results above, that a termination analysis based on abstract composition has higher precision than one based on abstract-relation transformers (as a trade-off for the higher complexity). In fact, one can always define a termination analysis based on abstract-relation transformers that has higher precision than one based on abstract composition, sometimes strictly higher.[3] We distinguish two distinct causes for the difference in precision.

- Both the abstract relation transformer $F^{\#}(a, \tau)$ and the abstract composition $a \circ^{\#} a_\tau$ define an abstraction of the relation $\gamma(a) \circ \rho_\tau$. However the former can be strictly more precise than the latter, since the abstract composition has to be an abstraction of a superset of $\gamma(a) \circ \gamma(a_\tau)$. In fact, there are cases of abstract-relation transformers $F^{\#}$ with a yes-answer (proving that the input program terminates) such that no abstract composition $\circ^{\#}$ exists with a yes-answer.
- A set of abstract relations X that contains all elements $a \circ^{\#} a_\tau$ where $a \in X$ can be strictly smaller than one that contains all elements $a_1 \circ^{\#} a_2$ where $a_1 \in X$ and $a_2 \in X$.

  Even if we require the abstract-relation transformers $F^{\#}$ to be defined by $F^{\#}(a, \tau) = a \circ^{\#} a_\tau$, there are cases where the LFP CHECKING FOR ABSTRACT RELATIONS returns a yes-answer but the CLOSURE CHECKING FOR ABSTRACT RELATIONS returns a no-answer.

Finally, a potential advantage of abstract composition above abstract-relation transformers. The latter can be defined and constructed only once the input program with its transitions τ is known. The former can be defined and constructed in a pre-processing step, once the set of abstract relations $\mathcal{T}^{\#}$ is fixed.

5.3 Special Case: Associative Composition of Abstract Relations

In this section we investigate the special case where the abstract composition $\circ^{\#}$ of abstract relations is associative. The example of size-change termination falls into this case, i.e., the composition of size-change graphs is associative. We will see that associativity has two consequences.

[3] We discuss the change of setting and the resulting differences in the online version of this paper [20].

- The decision problem Closure Checking for Abstract Relations can be reduced to the decision problem Lfp Checking for Abstract Relations . We thus obtain a better upper bound for the complexity.
- The decision problem can be further reduced to a decision problem where the inclusion in the question "$cl(\{a_\tau \mid \tau \in \mathcal{T}\}) \subseteq \mathsf{GOOD}$" is restricted to a subset of abstract relations. The subset consists of *idempotent* elements a, i.e., where $a \circ^\# a = a$. Thus, we can replace the input parameter GOOD by a subset of GOOD (containing idempotent elements only), and reserve the well-foundedness check for only those elements.

We recall that both of the above decision problems require the well-foundedness of every relation denoted by an abstract relation a in GOOD.

Theorem 54. *The closure of an* associative *abstract composition* $\circ^\#$ *equals the least fixpoint of the abstract-relation transformer* $F^\#$ *defined by*

$$F^\#(a, \tau) = a \circ^\# a_\tau$$

i.e.,

$$lfp(\{a_\tau \,|\, \tau \in \mathcal{T}\}, F^\#) = cl(\{a_\tau \mid \tau \in \mathcal{T}\}, \circ^\#).$$

Corollary 55. *If the abstract composition* $\circ^\#$ *is associative,* Closure Checking for Abstract Relations *is decidable in space* $\mathcal{O}(log^2|\mathcal{T}^\#|)$.

We note the correspondence to Theorem 11.

Theorem 56. *If every* idempotent *element in the closure* $cl(\{a_\tau \,|\, \tau \in \mathcal{T}\}, \circ^\#)$ *of an* associative *abstract composition, then* every *element (idempotent or not) denotes a well-founded relation.*

Proof. We show that whenever some element of $cl(\{a_\tau \mid \tau \in \mathcal{T}\}$ denotes a relation that is not well-founded, then $cl(\{a_\tau \mid \tau \in \mathcal{T}\}$ contains an idempotent element that denotes a non-well-founded relation.

Let $a \in \mathcal{T}^\#$ be an abstract relation. We define the following notation recursively for $n \geq 1$.

$$a^n = \begin{cases} a & \text{if } n = 1 \\ a^{n-1} \circ^\# a & \text{otherwise} \end{cases} \qquad \gamma(a)^n = \begin{cases} \gamma(a) & \text{if } n = 1 \\ \gamma(a)^{n-1} \circ \gamma(a) & \text{otherwise} \end{cases}$$

Since $(cl(\{a_\tau \mid \tau \in \mathcal{T}\}), \circ^\#)$ is a finite semigroup, $(\{a^n \mid n \geq 1\}, \circ^\#)$ is also a finite semigroup. By stepwise induction and definition of an abstract composition, we get that the inclusion

$$\gamma(a)^n \subseteq \gamma(a^n)$$

holds for $n \geq 1$.

A well-known result is that every finite semigroup has an idempotent element. Let $k \in \mathbb{N}$ be a natural number, such that a^k is idempotent. Assume the relation $\gamma(a)$ is not well-founded. Then the relation $\gamma(a)^k$ and its superset $\gamma(a^k)$ are also not well-founded. Hence the idempotent element a^k denotes a relation that is not well-founded. □

This proves a slightly stronger result than Theorem 56: a sufficient condition is associativity of the abstract composition on the elements of the closure.

In the decision problem we define below, one may obviously restrict the elements in the input parameter GOOD to idempotent elements.

Problem: Associative Closure Checking for Abstract Relations

Input:

- a program $P = (\Sigma, \mathcal{T}, \rho)$.
- a transition abstraction $(\mathcal{T}^{\#}, \gamma, \{a_\tau \mid \tau \in \mathcal{T}\})$.
- an *associative* abstract composition $\circ^{\#}$
- a subset $\mathsf{GOOD} \subseteq \mathcal{T}^{\#}$ such that every element of GOOD denotes a well-founded relation.

Property: $\{a \in cl(\{a_\tau \mid \tau \in \mathcal{T}\}) \mid a \text{ is idempotent}\} \subseteq \mathsf{GOOD}$

Example 57. In the setting of size-change termination, where the transitions τ are the calls, the abstract relations are the size-change graphs, the (associative!) abstract composition is the composition operator " ; " of size-change graphs, we choose GOOD to be the set of all *idempotent* size-change graphs G with an arc $z \xrightarrow{\downarrow} z$ (the denotation of G is then a well-founded relation).

Theorem 58 (Associative Closure Checking for Abstract Relations and Termination). *Program P is terminating if the answer to the decision problem* Associative Closure Checking for Abstract Relations *is yes.*

Proof. by Theorem 52 (or Theorem 54 together with Theorem 46) and Theorem 56. □

6 Discussion: Qualitative Differences

The research on concepts and methods based on size-change termination (SCT) resp. transition invariants (TI) involves somewhat different assumptions. All are, however, related to linear *computational paths* and to relations among *first-order values*. This is in contrast to other approaches, for example Gödel's higher-order primitive recursive functions, and analyses of higher-order programs studied among others by Bohr and by Sereni [24,36,37]. In this section, we discuss qualitative differences between SCT and TI.

Analysis principles. The SCT analysis traces flow of data in a well-founded data set between *single variables* over all of a program's transition sequences. It reports termination if every infinite transition sequence would cause an infinitely descending value flow between variables. A TI analysis, in contrast, focuses on showing that the program's overall state transition relation is well-founded; there is no a-priori known well-founded data set in which to trace program data flow.

SCT models are uninterpreted. SCT program data may be any well-founded set, not necessarily well-ordered and not fixed, e.g., to the integers or natural numbers. Thus SCT analysis cannot conclude, e.g., that $x < y$ implies $x+1 \leq y$. The TI frameworks do not explicitly mention a value domain, although practical tools (based on RankFinder) search for ranking functions over the (positive or negative) integers.

Intensionality/extensionality: size-change analysis is *intensional*: it works by manipulating not semantic objects themselves, but rather a priori determined syntactic objects that describe them: size-change graphs. TI analyses, in contrast, are formulated *extensionally*, in terms of direct manipulation of semantic values, i.e., binary relations on states. In practice, formulas in first-order logic are used to denote these relations.

Decidability: The size-change termination property is decidable, and. its complexity is understood. The calculations in the SCT analysis are done according to fixed combinatorial techniques known in advance: the definition of "$;$" and the recognition of in-place decreases $z \stackrel{\downarrow}{\rightarrow} z$.

In contrast, a TI analysis addresses an undecidable verification problem. As already mentioned, the very motivation behind the work in [31] was to carry over the ideas of [27] to verification methods in the style of *software model checking* [3,4].[4] A software model checker uses theorem provers and decision procedures as oracles that 'solve' potentially undecidable problems to implement *predicate abstraction* and *counterexample-guided abstraction refinement* (see [3,4]).

Parametrisation: SCT is a relatively rigid framework. It uses a generic set of building blocks to define size-change graphs for every program. Thus, for example, SCT never traces the flow of values that may increase. It is also insensitive to tests in the program being analysed.

In contrast, the starting point of the TI analysis based on transition predicate abstraction (TPA) is a set of predicates $\mathcal{P}$ that parametrizes the abstraction function α. The TI analysis, if used together with counterexample-guided abstraction refinement, requires (in addition to testing well-foundedness) the ability to compute a suitable approximation to the abstraction function α.

An example of reasoning based on abstraction: The generic class $\mathcal{P}_{SCT}$ captures the expressivity of size-change graphs. Inspection of $\mathcal{P}_{SCT}$ reveals that comparisons can be made only between a current variable value and a next variable value. Thus it is impossible to express relations between two current values, as would be required to model tests in the program being analysed.

Acknowledgements. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14

[4] By principle, software model checking, if based on predicate abstraction (or any other abstraction of a program to a finite-state system), is unable to prove termination of programs with executions of unbounded length.

AVACS). The second author thanks the Alexander von Humboldt-Stiftung for supporting a stimulating half-year stay at the Institut für Informatik at the University of Freiburg.

References

1. Avery, J.: Size-change termination and bound analysis. In: Hagiya and Wadler [19], pp. 192–207.
2. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI, pp. 203–213 (2001)
3. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In: Katoen and Stevens [25], pp. 158–172
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
5. Ben-Amram, A.M.: Size-change termination with difference constraints. ACM Trans. Program. Lang. Syst. 30(3), 1–31 (2008)
6. Ben-Amram, A.M.: A complexity tradeoff in ranking-function termination proofs. Acta Informatica 46(1), 57–72 (2009)
7. Ben-Amram, A.M.: Size-change termination, monotonicity constraints and ranking functions. Logical Methods in Computer Science (2010)
8. Ben-Amram, A.M., Codish, M.: A SAT-based approach to size change termination with global ranking functions. In: Ramakrishnan and Rehof [33], pp. 218–232
9. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. ACM Trans. Program. Lang. Syst. 29(1) (2007)
10. Ben-Amram, A.M., Lee, C.S.: Ranking functions for size-change termination II. Logical Methods in Computer Science 5(2) (2009)
11. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: Hofmann and Felleisen [22], pp. 211–224
12. Choueiry, B.Y., Walsh, T. (eds.): SARA 2000. LNCS, vol. 1864. Springer, Heidelberg (2000)
13. Codish, M., Lagoon, V., Stuckey, P.J.: Testing for termination with monotonicity constraints. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 326–340. Springer, Heidelberg (2005)
14. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Schwartzbach and Ball [35], pp. 415–426
15. Cook, B., Podelski, A., Rybalchenko, A.: Summarization for termination: No return! Journal of Formal Methods in System Design (2010)
16. Cousot, P.: Partial completeness of abstract fixpoint checking. In: Choueiry and Walsh [12], pp. 1–25
17. Glenstrup, A.J., Jones, N.D.: Termination analysis and specialization-point insertion in offline partial evaluation. ACM Trans. Program. Lang. Syst. 27(6), 1147–1215 (2005)
18. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL 2009: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 16–28. ACM, New York (2009)
19. Hagiya, M., Wadler, P. (eds.): FLOPS 2006. LNCS, vol. 3945. Springer, Heidelberg (2006)

20. Heizmann, M., Jones, N.D., Podelski, A.: Size-change termination and transition invariants (online version) (2010), http://swt.informatik.uni-freiburg.de/staff/heizmann/SCTandTI.pdf
21. Hinze, R., Ramsey, N. (eds.): Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, 2007, October 1-3. ACM Press, New York (2007)
22. Hofmann, M., Felleisen, M. (eds.): Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19. ACM, New York (2007)
23. Jones, N.D.: Computability and Complexity from a Programming Perspective. In: Foundations of Computing, 1st edn., MIT Press, Boston (1997)
24. Jones, N.D., Bohr, N.: Call-by-value termination in the untyped lambda-calculus. Logical Methods in Computer Science 4(1) (2008)
25. Katoen, J.-P., Stevens, P. (eds.): TACAS 2002. LNCS, vol. 2280. Springer, Heidelberg (2002)
26. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
27. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL 2001: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 28, pp. 81–92. ACM Press, New York (2001)
28. Mannaand, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Springer, Heidelberg (1995)
29. Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.): The Essence of Computation; Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS, vol. 2566. Springer, Heidelberg (2002)
30. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen and Levi [38], pp. 239–251
31. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, pp. 32–41. IEEE Computer Society, Los Alamitos (2004)
32. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL 20505: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 32, pp. 132–144. ACM, New York (2005)
33. Ramakrishnan, C.R., Rehof, J. (eds.): TACAS 2008. LNCS, vol. 4963. Springer, Heidelberg (2008)
34. Schmidt, R.A. (ed.): CADE-22. LNCS, vol. 5663. Springer, Heidelberg (2009)
35. Schwartzbach, M.I., Ball, T. (eds.): Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14. ACM, New York (2006)
36. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In: Hinze and Ramsey [21], pp. 71–84
37. Sereni, D., Jones, N.D.: Termination analysis of higher-order functional programs. In: Yi [40], pp. 281–297
38. Steffen, B., Levi, G. (eds.): VMCAI 2004. LNCS, vol. 2937. Springer, Heidelberg (2004)
39. Swiderski, S., Parting, M., Giesl, J., Fuhs, C., Schneider-Kamp, P.: Termination analysis by dependency pairs and inductive theorem proving. In Schmidt [34], pp.322–338
40. Yi, K. (ed.): APLAS 2005. LNCS, vol. 3780. Springer, Heidelberg (2005)

Using Static Analysis in Space: Why Doing so?

David Lesens

EADS Astrium Space Transportation, Route de Verneuil, 78133 France
david.lesens@astrium.eads.net

Abstract. This paper presents the point of view of an industrial company of the space domain on static analysis. It first discusses the compatibility of static analysis with the standards applicable for the development of critical embedded software in the European space domain. It then shows the practical impact of such a technology on the software development process. After the presentation of some examples of industrial use of static analysis, it concludes by envisaging the future needs of industry concerning static analysis.

Keywords: Static analysis, industrial space software.

1 Introduction

The use of static analysis is today recognized by the research community as a major potential contributor in the improvement of software engineering.

However, by nature, a commercial company is not philanthropic. It will invest in a static analysis tool (and potentially on research programs aiming at developing a static analysis tool, if existing tools do not fulfill the needs) only if it has a positive return on investment.

The objective of this paper is to present the point of view of an industrial company (whose field is the spatial activity) about static analysis:

- Why is static analysis important for critical embedded system? (surprisingly, the answer may be not so obvious considering the constraints imposed by the applicable standards)
- What is the current use of the static analysis in the European space domain?
- What are the possible prospects of static analysis in space applications, i.e., what are the expectations of the industrial world towards the academic one?

Generally, the practical reasons pushing the use of a technology may be ones of the following:

- The development of the commercialized product (e.g. a space launcher) is not possible without this technology. This is for instance the case for the propulsion system (main driver of the launcher performance). Closer for the computer science, a correct robustness of the onboard processor with respect to its physical environment (vibration, temperature, radiation...)

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 51–70, 2010.
© Springer-Verlag Berlin Heidelberg 2010

is mandatory. Considering the software itself, static analysis is not really an unavoidable technology (from years a lot of launchers are reaching space without using it)

- The technology is imposed by the customer. For the general public, this requirement may for instance be dictated by a fashion or the state of the art (people prefer MP3 players, even if compact disks or vinyl records are in fact of much higher quality). In the industrial domain, the state of the art is described by the applicable standards. But if a standard may impose the use of a technology, it may as well forbid it (or just authorize it without imposing it).
- The technology allows a quantifiable gain for the industrial company in term of costs or duration of the development. By using a static analysis tool, an industrial company may wish to decrease its testing effort with the preservation of the same level of quality for its product. This financial gain brought by static analysis may be not straightforward, especially in the case where the applicable standards do not allow for instance replacing a complete V&V (Verification and Validation) activity by a static analysis.
- The technology allows a quantifiable gain for the industrial company in term of quality of the commercialized product. This criterion is not always the major one in some domains. Indeed, it may be preferable (from a pure short term commercial point of view) to be the first provider of a new product (for instance a new smart-phone) with a medium quality than to be the second one on the market, even with a higher quality. A company may even not survive if its product is available too late. However, on other domains, a failure of the system (annoying but generally tolerable for instance for a smart-phone if it does not appear to often) may be catastrophic in term of human life (e.g. for an aircraft or a nuclear plant), or in term of financial loss (e.g. for an unmanned space launcher).

These criterions are often summarized by the formula "*On Time, on cost, on quality*".

This paper is organized in the following manner. After this introduction, Sect. 2 analyzes the compatibility between static analysis and the standards applicable in the European space domain. Section 3 analyses more precisely the practical impact of static analysis on the development. Section 4 presents some feedbacks of use (in operational or research projects) of static analysis as performed at Astrium Space Transportation. Finally, Sect. 5 will conclude by some perspectives and some orientation wishes to the research community.

2 Static Analysis and the European Standards for Space

The objective of this section is to analyze the compatibility between an industrial standard (in our case the European standard for space) with an approach involving static analysis. The results of this analysis may certainly differ depending on the industrial used standards, but we assume that it will be only in a light manner.

2.1 Principles of the ECSS

The European space industry has defined in the scope of ESA (the European Space Agency) its own standards of development: the ECSS (European Committee for Space Standardization). The two main norms concerning the software are the ECSS-E-ST-40C (Space engineering – Software) and the ECSS-Q-ST-80C (Space product assurance – Software product assurance). In this section, we will reference the versions [fSS09].

These standards, even if specific for space, do not of course completely differ from other standards for the development of critical software (such as the DO178B or C for civil airborne systems), the expertise being equivalent. We may roughly distinguish three types of activities:

– the development itself (composed by the elicitation of software related system requirements and of the software requirements, by the software architecture, by the design and by the code implementation)
– the verification[1] ("*to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and input*") which is formalized by several reviews (system requirements review, preliminary design review, critical design review, qualification review, acceptance review)
– and finally the validation ("*to confirm that the requirements baseline functions and performances are correctly and completely implemented in the final product*")

These main activities are completed by the specific cases of reuse of an existing piece of software and by the maintenance of the software.

Figure 1 depicts this development cycle, compatible with the classical V model of development.

The next paragraphs of this section will discuss the potential contributions of static analysis to these activities: mainly validation and verification but also ISVV (Independent Software Verification and Validation) and software reuse. The impact of static analysis on the development phase itself is not direct. However, the section 3 will discuss indirect impacts on the programming languages or modeling languages choices.

2.2 Static Analysis for Validation

The paragraph 5.6.3.1.b of the ECSS-E-ST-40C specifies the following requirement: "*Validation shall be performed by test*". This simple requirement may cancel one of the most important arguments for the use of static analysis: decreasing the testing effort with the preservation of the same level of quality for

[1] A reader more familiar with airborne systems will notice that this ECSS definition differs from the DO178 definition. What is called "*verification*" by the ECSS is called "*validation*" by the DO178 and vice-versa.

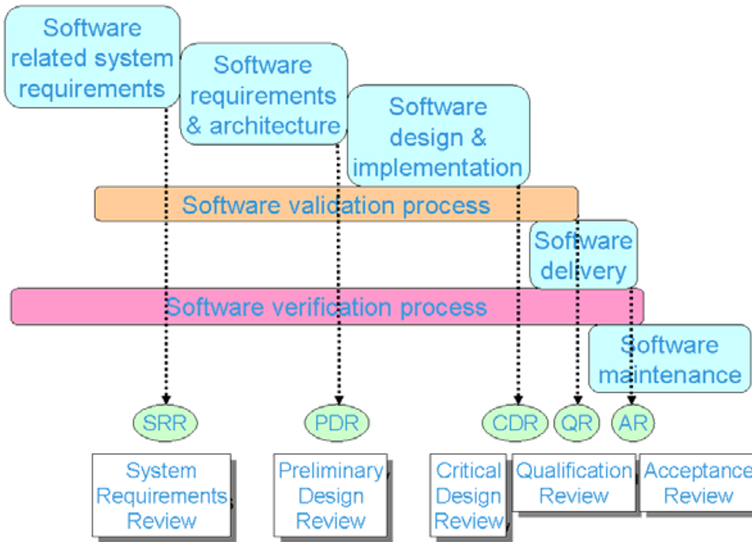


Fig. 1. The Software development cycle according to the ECSS

the product. This argument may be true for some industries, but not for European space. Even if the industrial company is convinced that some tests may be suppressed thanks to static analysis (and suppressing some tests is important for reducing the costs), it is a priori not allowed by the standard.

This restriction is of course justified by the need to explicitly validate the final software product, avoiding the possible errors due to the compilers and / or hardware design bugs. Thus, for instance for a piece of software with a criticality level A (the highest level of criticality), the object code coverage on the real hardware is required (paragraph 5.8.3.5.e of the ECSS-E-ST-40C).

However, the next paragraph of the standard (5.6.3.1.c) nuances the previous requirement: "*If it can be justified that validation by test cannot be performed, validation shall be performed by either analysis, inspection or review of design*". The good news (concerning the use of static analysis) is that alternative methods to testing may be used, on the condition that "*it can be justified that validation by test cannot be performed*".

We have thus to more deeply analyze the validation tests which are required on space software. In addition to the classical functional tests in front of each requirement of the software technical specification, the paragraph 5.6.3.1.a.1 requires "*test procedures including testing with stress, boundary, and singular inputs*" (Stress tests "*evaluate [...] a software component at or beyond its required capabilities*").

Can static analysis help validating functional requirements? Certainly yes! The section 4.3 shows an example of use of model checking applied to the safety software of a spacecraft, which has allowed detecting errors before starting the test campaign.

As this validation is globally achievable with tests, the ECSS forbids replacing completely tests by static analysis. The main validation mean remains test (in order to validate at the same time the code, the compiler and the hardware). Static analysis, in this case, may be considered as an additional not too expensive activity increasing the confidence in the software (i.e. avoiding extra stress the day before launch within the software development team).

Stress ("*evaluating [...] a software component [...] beyond its required capabilities*") is an activity which may be less natural than functional testing in the sense that the software shall be evaluated robust to preposterous inputs: What is supposed to be the behavior of my software in case of inputs aiming at the estimation of a negative altitude for a spacecraft? In this case, a reasonable expectation would be the absence of run time errors: the part of the software behaving strangely (e.g. computation of a negative altitude) shall not prevent another part of the software behaving correctly (e.g. a telemetry warning the ground control centre of the suspicious altitude computed by the software).

In this case, the large amount of possible preposterous may prevent any efficient testing or review. Abstract interpretation may be then a good complement to classical testing.

The section 4.2 shows such examples of application of abstract interpretation on space software.

2.3 Static Analysis for Verification

Among the different criterions to be checked during the verification activity, the paragraph 5.8.3.5 of ECSS-E-ST-40C highlights the following points (among others):

- "*The code implements safety, security [...]*"
- "*The effects of run-time errors are controlled*"
- "*There are no memory leaks*"
- "*Numerical protection mechanisms are implemented*"
- "*The supplier shall verify source code robustness (e.g. resource sharing, division by zero, pointers, run-time errors)*"

But it is well known that "*program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence*" [Dij88]. This is particularly the case from the above mentioned list of bugs (safety, security, run-time errors, memory leaks, numerical protection). These verifications are thus supposed to be partially covered by reviews, which are well known to be very costly (human resource is always costly) and not efficient (and also very boring for the code reviewer, even if this argument is rarely taken into account by the managers).

That is why ECSS-E-ST-40C recommends "*using static analysis for the errors that are difficult to detect at runtime*" (paragraph 5.8.3.5.f).

We can add to this list the proof of absence of loss of numerical accuracy (ECSS-Q-ST-80C, paragraph 7.1.7: "*Numerical accuracy shall be estimated and verified*") which is also a particularly difficult task. Figure 2 shows a classical example proposed by P. Cousot. Depending of the values of x and a, the value of $(x + a) - (x - a)$ can be very different from $2a$.

```
double x, a; float y, z;
x = 1125899973951488.0;
a = 1.0;
y = ( x+a ); z = ( x-a );
printf( "%f, ", y-z );
// Computing result = 134217728.000000
x = 1125899973951487.0;
a = 1.0;
y = ( x+a ); z = ( x-a );
printf( "%f", y-z );
// Computing result = 0.000000
```

Fig. 2. Example of loss of numerical accuracy

Section 4.2 presents a result achieved with the Fluctuat tool on space software.

2.4 Static Analysis for ISVV

In order to increase the confidence in the space systems, the main customer for space products in Europe (the European Space Agency) requires an ISVV (Independent Software Verification and Validation). Independent means generally a team external to the contractors (or at least a team distinct from the development team).

The ECSS-Q-ST-80C requires (paragraph 6.2.6.13) that

- "*Independent software verification shall be performed by a third party.*"
- "*Independent software verification shall be a combination of reviews, inspections, analyses, simulations, testing and auditing.*"
  "*(This requirement is applicable where the risks associated with the project justify the costs involved)*".

The activity of ISVV differs largely from the classical V&V (Verification and Validation) activity because the involved people have no a priori knowledge of the product. It may thus be very difficult for the ISVV team to bring a real added value to the classical V&V, for which the involved people have normally a perfect mastery. That is why the ISVV activity often focuses itself in practice (even if this focus is not required by the standard) on specific points such as for instance the verification of absence of buffer overflowing or of data going out of

range. These activities are especially well adapted to be treated by automatic static analysis tool. Indeed, using static analysis for the detection of run-time errors would free human resource for activities with more added values.

2.5 Static Analysis for V&V of Reuse Software and Regression Testing

Reusing an existing piece of code is often considered as a natural way to save money (why not reusing a piece of code that has previously proved to work correctly?). However, for critical software, reuse shall be carefully performed to avoid any risk of inconsistence between reused and specifically developed code. Indeed, a classical error consists in reusing a piece of code in a lightly different context which makes the resulting system faulty (even if the initial piece of code in its initial context was perfectly valid). The ECSS-Q-ST-80C (paragraph 6.2.7.8.a) requires thus that "*Reverse engineering techniques shall be applied to generate missing documentation and to reach the required verification and validation coverage*".

In this case, as well as in the case of regression testing ("*selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements*"), static analysis can be an important help because it is automatic and exhaustive.

2.6 Conclusion on Static Analysis and Standards

A detailed analysis of the ECSS standards shows that the use of static analysis is naturally accepted by the standard ... if it does not imply the suppression any other V&V activities. Except for some specific cases (ISVV, software reuse), static analysis can of course be used as a way to improve the confidence on the software, but without a direct reduction of cost achievable by the suppression of testing. Improvement of the quality, yes; decrease of the direct costs, no!

3 Impact of Static Analysis on the Development Strategy

3.1 Link between Static Analysis and Development Strategy

After a check that static analysis is compatible with the applicable standards for the domain (ECSS for space, DO for airborne systems, etc.), an industrial company may decide to use it. According to the analysis performed in the previous section (Sect. 2), the added value of static analysis seems to be in the V&V activities.

However, the context of application of static analysis may have a strong impact on the return on investment:

- Use of static analysis for ISVV or acceptance of the software: The software has already been validated and it is not possible any more to modify it. The static analysis technique shall then be able to deal with the used programming language, design choices and coding style.
- Introduction of static analysis at the end of the development: The software has been developed without taking into account the constraints needed by a potential static analysis tool. However, before starting the formal validation, the verification team proposes to use a static analysis tool. It is still possible to perform light modification to the software (by adding some assertions in the code for instance), but it is of course to late to modify the major choices of technology for the development.
- Introduction of static analysis at the beginning of the development: Considering that static analysis will facilitate the V&V, it is decided to take into account from the beginning the associated constraints. The gamble is made that the induced extra costs (if any) will be compensated by the future savings (i.e. that the return on investment will be positive). In this case, it is possible to select the best adapted modeling or programming language, design patterns and coding rules.

This context can barely be controlled by the contractor: if a company is required to perform an ISVV, it has not the responsibility for the software development; when a reuse policy is used, the major design choices are imposed[2].

This section discusses the ideal case when the software development starts from scratch (from a blank page). Do the design choices have an impact on the efficiency of static analysis tools?

3.2 Static Analysis and Programming Language

Static analysis covers a wide range of techniques. The minimal analysis on any piece of software, even the less critical ones, is of course performed by the compiler, which checks the syntax and the semantics of the programming language. The level and thus interest of this minimal static analysis depends naturally on the semantics of the selected programming language. If this assertion may be considered very trivial from a scientific point of view, it may be of major importance for an industrial development.

Let us for instance compare through a couple of examples the C programming language[3] and the Ada programming language[4].

---

[2] This is for instance the case for the development of Ariane 5 Mid Life Evolution. In order to upgrade the upper stage of the launcher, the software controlling the lower stage and developed in the 90's shall be reused. This is a strong constraint on the choice for a new Static Analysis tool.

[3] C is today the language the most widely used for critical embedded system; the use of Java is more important for non critical embedded system but is today very limited for critical ones.

[4] Ada is a language specialized for critical embedded software.

```c
// An incorrect C program
// compiles without warning
typedef enum { ok, nok } t_ok_nok;
typedef enum { off, on } t_on_off;

void main () {
    t_ok_nok status = nok;

    if( status == on ) {
        lift_off();
    }
}
```

```ada
-- An incorrect Ada program
-- does not compile
type t_ok_nok is ( ok, nok ) ;
type t_on_off is ( off, on ) ;

procedure Main is
    status : t_ok_nok := nok;
begin
    if status = on then
        lift_off;
    end if;
end Main;
```

Fig. 3. Incorrect C and Ada programs. The C program is accepted by the C compiler while the Ada version is rejected by the less permissive Ada compiler.

In the classical example of Fig. 3, two types (*ok/nok* and *on/off*) have been defined. The programmer has mixed up the two types: the status of type *ok/nok* is compared to *on*, which unfortunately has the same numerical value than *nok*. The C program is accepted by the C compiler while the Ada version is rejected by the less permissive Ada compiler.

The next more subtle example (Fig. 4) is extracted from [Bar03].

```c
// An incorrect C program
// compiles without warning
light = red ;
if ( light == green );
{
    Lift_off() ;
}
```

```ada
-- An incorrect Ada program
-- does not compile
light := red ;
if light = green
then;
    Lift_off() ;
end if;
```

Fig. 4. Incorrect C and Ada programs. The C program is accepted by the C compiler while the Ada version is rejected by the less permissive Ada compiler.

The error comes from the semi-colon at the end of the condition. It is detected by an Ada compiler but not directly by a C compiler (external syntax verification tools may detect this error).

In these two examples, if the software decides the lift-off of a launcher (authorized if the status is ok or if the light is green), the software developed in Ada will safely abort the flight, whereas the software developed in C will authorize the flight with possible very expensive consequences if either the launcher or its launch pad are destroyed due to the default software.

More generally, the strongest the semantics of the programming language is, the largest will be the number of errors detected by the compiler, and the MORE efficient static analysis can be.

The question which can be raised is then the following: Is it really useful to spend a lot of effort and money to develop tools able to statically analyze a programming language which is intrinsically subject to errors (such as C)? Or shall we concentrate our efforts on the static analysis of "clean" programming language (such as Ada)?

The answer to this question is not so straightforward first because even a programming language with a strong semantics (such as Ada) can be in some cases ambiguous and difficult to analyze. The example of Fig. 5 proposed by [Cha01] shows an example of Ada code containing a function with a side effect.

```
procedure Side_Effect is
   X, Y, Z, R : Integer;
   function F (X : Integer) return Integer is
   begin
      Z := 0;
      return X + 1;
   end F;
begin
   X := 10;
   Y := 20;
   Z := 10;
   R := Y / Z + F (X); – order dependency here
   – R = 13 if L then R evaluation,
   – constraint error if R then L
end Side_Effect;
```

Fig. 5. Function with a side-effect. Depending of the order of evaluation, the program may raise a constraint error.

Because of this side effect, the behavior of the program depends on the order of evaluation of an addition. We can then imagine a successful test on a host machine (a work station) and an incorrect one on the target machine (the embedded processor). In order to avoid such problems, a possible approach would be to simply forbid the use of side effect on a function (this is for instance the choice of the SPARK Ada programming language), which may be a quite strong constraint to the developer.

The second reason explaining the difficulty to choose a programming language is that in a perfect world (i.e. a world where the programming language would be selected only for its intrinsic qualities), it would be completely useless to try developing more complex static analysis tools for programming languages with a weak semantics; it would be sufficient (even if not trivial) to develop static analysis tool only for programming languages with a strong semantics. But in practice, one may select its programming language for other (very good)

reasons than intrinsic technical ones (such as the long term availability of the technology, the availability of the compiler for the language and the hardware target, performances restrictions (CPU, memory), request from the customer).

3.3 Static Analysis and Model Driven Engineering

The use of modeling techniques for describing on-board and ground space systems has been identified by many studies as an appropriate way to master the complexity of software. Today, projects show an increasing interest for applying a model-based approach at different levels:

- Representation of the software or system architecture (e.g. with AADL, SysML, etc.);
- Representation of the software or system behavior (e.g. with SCADE, Simulink, SystemC, etc.);
- Representation of the implementation of the software and of the system (e.g. with SCADE, UML, VHDL, etc.).

The use of a graphical model to describe in a more or less abstract way a piece of software or more generally a system has two main advantages:

- The graphical syntax of the model may be more intuitive than a textual one and may then greatly improved the communication between the development teams (for instance between a system team, composed by non software expert, and the software team)
- The strong semantics of the modeling language may greatly facilitate static analysis

Static Analysis of Static Architecture. Some trivial analysis can for instance be applied on class diagrams of HOOD (Hierarchic Object-Oriented Design) or of UML (Unified Modeling Language) for interface checking. These trivial analyses shall not be forgotten because the simplest analyses offer often the highest return on investment.

However, even for some trivial analysis such as data flow analysis, the semantics of the modeling language has a major impact on the interest of this analysis for the developer.

Let us consider for instance the two examples of structure diagrams, respectively described in SysML (Fig. 6) and in SCADE (Fig. 7):

These two models represent the same exchange of information between two software blocks. SysML (respectively SCADE) is a modeling language intended to be used at system level (respectively at software level). The system description shown Fig. 6 is perfectly valid. But the same description at software level (Fig. 7) becomes not valid because there is an ambiguity about the order of execution of the two blocks. The corrected SCADE model (which can generate code) is shown on Fig. 8).

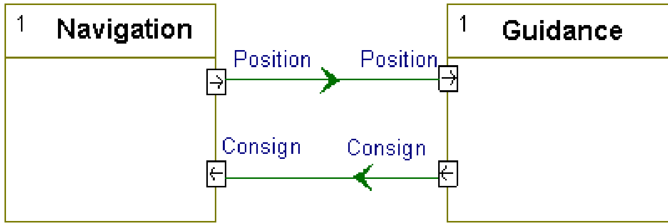


Fig. 6. The architecture of a piece of software modeled in SysML diagram example

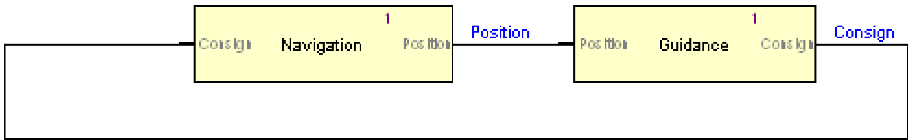


Fig. 7. The architecture of a piece of software modeled in SCADE diagram example

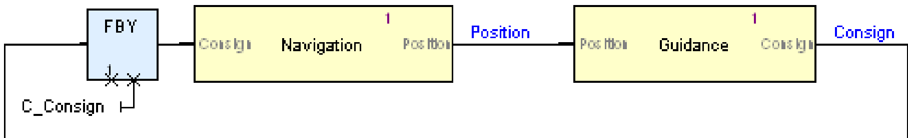


Fig. 8. The corrected SCADE model of the architecture

Static Analysis of Dynamic Behavior. Next to the analysis of the static architecture of the model one is the analysis of the dynamic behavior of this model. Here as well, the choice of the modeling language has a strong impact on the results we can expect from a static analysis. This will be illustrated with a single example (Fig. 9).

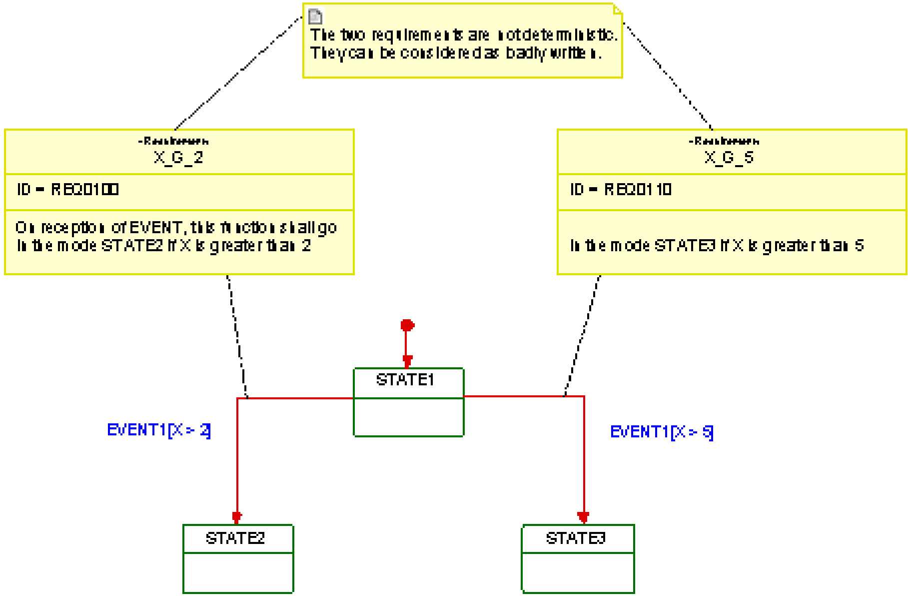


Fig. 9. An ambiguous UML model

This model is indeterminist, which can imply non predictable behavior (see Fig. 10). The two diagrams (on the left and on the right) modeled in UML are strictly equivalent (from a graphical point of view). However, the simulations of these two models with the Rhapsody tool (from IBM Software) will give different results:

- for the diagram on the left, the transition guarded by $x > 2$ is triggered,
- whereas for the diagram on the right, the transition guarded by $y > 5$ is triggered.

This model is valid at system level (all the details of implementation have not yet been frozen), but not at software level. For instance, the ECSS standards for critical software require deterministic testing (tests which can be replayed); and we naturally expect that the software will behave in operation as it has behaved during test. For the generation of code for a piece of critical software, using an indeterminist SysML is just not acceptable.

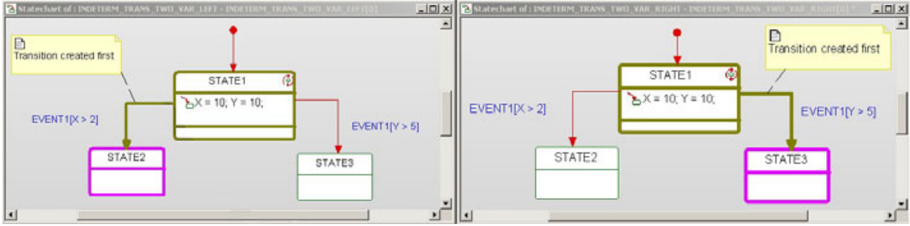


Fig. 10. An non-deterministic simulation

The experiment on static analysis presented in Sect. 4.3 will concern system analysis model. Static analysis can indeed help controlling the indeterminism. However, static analysis does not allow directly suppressing this indeterminism.

3.4 Conclusion on the Impact of Static Analysis on the Development Strategy

This section has shown that some choices in the development process (choice of a modeling language with automatic code generation, or choice of a programming language) have a direct potentially strong impact on the quality of the final software product and on the costs. Moreover, these choices directly impact:

- The quality of the results we can expect from a static analysis
- The difficulty to develop a tool implementing this static analysis

From a pure technical point of view, the choices of development method for critical embedded systems should be oriented toward the most adapted languages, i.e. with a strong semantics (such as SCADE or SPARK Ada). The development of static analysis tool should thus also be oriented toward these languages. From a more pragmatic point of view, it must unfortunately be noticed that for instance SysML or C are widely used and may have thus potential better long term availability than respectively SCADE or SPARK Ada. And static analysis tools are of prime importance to mitigate the weak semantics of such languages.

4 Static Analysis at Astrium Space Transportation

Astrium Space Transportation is one of the main users and supporters of static analysis tools in the European space domain. Now that the impacts of the standards on static analysis and the best design choices facilitating this static analysis have been presented in the previous sections, this section provides some (non exhaustive) examples of operational uses of static analysis and researches on static analysis at Astrium Space Transportation. They illustrate 3 topics: type checking, abstract interpretation and model checking.

4.1 Type Checking

In order to benefit from the most advanced verifications a compiler can provide, Astrium Space Transportation has used the Ada programming language since more than 20 years.

All the flight software benefit from the strong type checking the language provides. An important number of potential errors possible with other programming languages (such as C) are just non conceivable (see example of Fig. 4).

The capability of Ada to perform run-time checks (in order to verify for instance scalar out of specified bounds) is used during some testing activities (even if not used directly during the flight itself for performance reasons).

An attempt to reinforce the type checking of Ada to avoid dangerous construction is described Sect. 4.4.

Fig. 11. Ariane 5 lift-off (© ESA-CNES-ARIANESPACE)

4.2 Abstract Interpretation

Polyspace Verifier for the Detection of Run Time Errors. In order to increase the confidence Ada can intrinsically bring to the developer on the software quality, Astrium Space Transportation has collaborated on one of the first abstract interpretation tool: IABC (developed at INRIA). Its first assessment on the Ariane 5 flight software has lead to the creation of the Polyspace Technologies company (now TheMathworks) and of the commercial tool Polyspace Verifier.

Abstract interpretation is now used since more than 10 years in order to automatically detect potential run-time errors on several Astrium Space Transportation operational projects, such as Ariane 5 or the ARD ([LMR+98]).

The ARD (Atmospheric Reentrance Demonstrator) is an “Apollo like” Capsule which has been launched in 1998 by the Ariane 5 launcher in order to demonstrate the mastering of reentry techniques in Europe. After a sub-orbital trajectory with an apogee of 830 km, the ARD performed a 20 minutes atmospheric reentry, starting from a velocity of 27000 km/h to a final splashdown with a velocity of less than 20 km/h and a precision of impact less than 2 km before parachute opening (for a targeted precision of 5 km) .

Fig. 12. Atmospheric Reentry Demonstrator (© ESA)

The whole flight was under the control of the flight software (about 40,000 lines of ADA code). The Polyspace verifier tool detected 568 potential errors and proved automatically that 81% of them are false alarms. With the definition of specific coding rules (aiming at helping the static analysis without degrading the software performances), the Polyspace Verifier tool was able to automatically prove 90% of the potential errors. 64 errors remained to be manually proved, which remains within an acceptable scope in terms of developer work load.

Astrée for the Detection of Run Time Errors. One major drawback of Polyspace Verifier is the important number of false alarms (orange errors) which can be raised by the tool on a very complex piece of code. In collaboration with ENS, the Astrée tool has been assessed on the safety software of the ATV (Automated Transfer Vehicle).

Launch in March 2008 by a special version of the Ariane 5 launcher, the ATV (Automated Transfer Vehicle) docked itself to the ISS (International Space Station) less than a month later, with an extraordinary precision and control. In the meantime, the ATV has performed several demonstration approaches in order to prove than this 20 metric tons vehicle could get into the ISS vicinity without putting the station crew at risk.

The use of Polyspace Verifier on the ATV safety software was abandoned due to the too important number of false alarms (and replaced by a manual analysis). In the scope of a research project ([BCC+09]), Astrée was applied on the C code generated from a SCADE model developed by retro-engineering from the initial Ada code. After some updates of the model (especially the addition of a minimal set of numerical protection), the tool proved the total absence of run-time error.

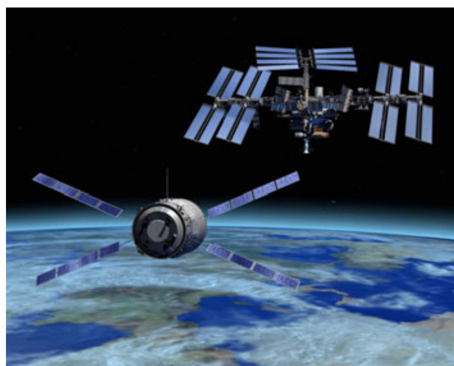


Fig. 13. Automated Transfer Vehicle (ATV) approaching the ISS (© EADS Astrium / Silicon Worlds)

However, Astrée is solely able to analyse C code, not Ada one. Thus, for the time being (and as long as a version of Astrée for Ada is not available), whatever the limitation of the Polyspace Verifier tool can be, it is still worth using it. As long as the perfect abstract interpretation tool will not be available, Astrium Space Transportation will still use Polyspace Verifier, even if not sufficient.

Fluctuat for the Analysis of Numerical Accuracy. Numerical accuracy is of vital importance for the safety of a spacecraft (see Sect. 2). In the same research project [BCC+09], Fluctuat has been assessed in collaboration with CEA.

This study has shown that embedded space software are difficult to analyze, due to non linearity (mainly in quaternion computation) and to complex filters (8th order filter in the case of the ATV safety software). In spite of these difficulties:

– Fluctuat has confirmed the absence of default issued from manual analyses
– Fluctuat has also delivered additional results on global errors which were not manually achievable.

4.3 Model Checking

Omega for the Verification of Functional System Properties. The deployment of the SysML modeling language for the capture of system requirements allocated to the software is in progress at Astrium Space Transportation. The benefits of SysML are very important because it facilitates the communication between teams with different fields of expertise:

– The team designing the spacecraft system (generally without expertise on embedded software), which is responsible to deliver the computing needs
– The team developing the embedded software, fulfilling the system needs

However, SysML has an important drawback (additionally to the complexity of SysML, which can be mastered by defining strict modeling guidelines, as described in [HM10]): the language is not formal (it is for instance unable to master indeterminism, as shown in Fig. 10). In collaboration with IRIT and Verimag, Astrium Space Transportation is currently studying the adaptation of the Omega profile ([BML01]) to SysML in order to improve the quality of the system requirements allocated to the software (thanks to a stronger formal semantics and to formal proof).

SCADE Prover for the Verification of Functional Software Properties. Model checking has been used by Astrium Space Transportation on the ATV program [Les01].

The level of criticality of the software controlling the ATV mission is C according to the ECSS-Q-ST-80C (major mission degradation), but the level of criticality of the software ensuring the safety of the ISS is A ("*loss of an interfacing manned flight system*"). In order to ensure the highest possible level of quality for this particular software, Astrium Space Transportation has modeled the system requirements allocated to the software in the SCADE modeling language. Then, the most critical functional properties (for instance "*in case of failure of the nominal system, the safety system takes eventually the control of the spacecraft*") have been formally and exhaustively verified by a proof engine.

4.4 Theorem Proving

In order to definitively prevent the errors due to some weakness of a programming language (such as the ones described in Fig. 4 and 5), Astrium Space Transportation is currently assessing the SPARK language.

SPARK is an annotated subset of ADA with some specific properties that are designed to make static analysis both deep and fast. The annotations take the form of special comments which are ignored by an ADA compiler but have semantic meaning for SPARK support tool, the SPARK Examiner. Annotations range in complexity from the description of data flows via global variables through to full pre- and post-condition predicates suitable for the formal verification of operations. A key property of SPARK is its complete lack of ambiguity. The language rules of SPARK, enhanced by its annotations, ensure that a source text can only be interpreted in one way by a legal ADA compiler. Compiler implementation freedoms such as sub-expression evaluation order, cannot affect the way object code generated from a SPARK source behaves. For example, a complete detection of parameter and global variable aliasing ensures that SPARK parameters have pass-by-copy semantics even if the compiler actually passes them by reference. The removal of ambiguous language constructs allows source-based static analysis of great precision and efficiency. Instead of using static analysis techniques to look for errors we can use them to prove the absence of certain classes of errors. This rather small linguistic difference masks a hugely-significant practical difference: only by eliminating ambiguity can we reach the goal of constructive, rather than retrospective, software verification.

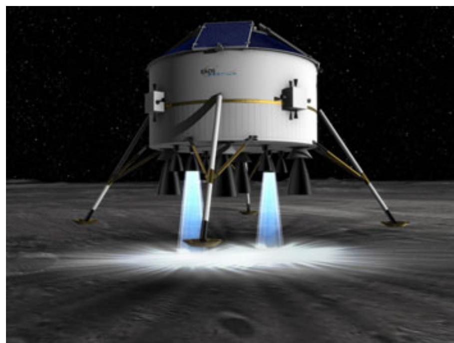


Fig. 14. Moonlander (© EADS ASTRIUM)

In parallel, Astrium Space Transportation is participating to the Hi-Lite project ([HL]) in order to combine the benefits of theorem proving, abstract interpretation and testing.

5 Conclusion

This paper has presented the point of view of an industrial company of the space domain about static analysis. It has especially tried to highlight the specific constraints an industrial company may have (such as applicable standards or long term availability of technologies). Some uses of static analysis at Astrium Space Transportation have also been presented (uses in operational projects or only in research projects).

In the future, Astrium Space Transportation will develop spacecrafts (new Ariane generation, space tourism, moon or Mars lander, see Fig. 14) with new challenges:

- More autonomous system: auto-diagnostic, FDIR (Fault Detection, Isolation and recovery)
- More critical software (rely on the software for safety concern, manned flight)
- Mastered costs (decrease of space budget dedicated for space)

For these projects, the classical V&V (mainly testing and review) required by the ECSS standards will not be sufficient (even if always mandatory). Static analysis will be for instance useful:

- To prove the absence of run-time error
- To prove the correct accuracy of numerical computation
- To evaluate the WCET (Worst Case Execution Time) for multi-threading architecture and hardware with cache-memory (and potentially multi-core)
- To meet the RAMS (Reliability, Availability, Maintainability and Safety) requirements
- To exhibit the correctness of functional properties (at both software and system levels)

Static analysis tools shall be adapted to the technologies of tomorrow (too permissive languages such as C should be definitively ruled out) and will need to be compatible, in order to benefit from the synergy between reviews, testing, formal method, abstract interpretation, theorem proving and model checking.

References

[Bar03] Barnes, J.: High Integrity Software. In: The SPARK Approach to Safety and Security, Addison Wesley, Reading (2003)

[BCC+09] Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Ghorbal, K., Goubault, E., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., Turin, M.: Space software validation using abstract interpretation. In: Data System In Aerospace (DASIA 2009), Istambul, Turkey (May 2009)

[BML01] Bozga, M., Mounier, L., Lesens, D.: Model checking ariane-5 flight program. In: Formal Methods for Industrial Critical Systems, FMICS 2001, Paris, France (July 2001)

[Cha01] Chapman, R.: Spark and abstract interpretation - white paper (2001)

[Dij88] Dijkstra, E.W.: On the cruelty of really teaching computing science, The University of Texas at Austin, USA (1988)

[fSS09] European Committee for Space Standardization. Ecss-e-st-40c and ecss-q-st-80c (March 2009)

[HL] Hi-Lite. Hi-lite project, http://www.open-do.org/projects/hi-lite/

[HM10] Hiron, E., Miramont, P.: Process based on sysml for new launchers system and software developments. In: Data System In Aerospace, DASIA 2010, Budapest, Hungary (June 2010)

[Les01] Lesens, D.: Use of the formal method scade for the specification of safety critical software for space application. In: Data System In Aerospace, DASIA 2001, Nice, France (May 2001)

[LMR+98] Lacan, P., Monfort, J.N., Ribal, L.V.Q., Deutsch, A., Gonthier, G.: The software reliability verification process. example of ariane 5. In: Data System In Aerospace, DASIA 1998 (1998)

Statically Inferring Complex Heap, Array, and Numeric Invariants

Bill McCloskey[1], Thomas Reps[2,3,⋆], and Mooly Sagiv[4,5,⋆⋆]

[1] University of California; Berkeley, CA, USA
[2] University of Wisconsin; Madison, WI, USA
[3] GrammaTech, Inc.; Ithaca, NY, USA
[4] Tel-Aviv University; Tel-Aviv, Israel
[5] Stanford University; Stanford, CA, USA

Abstract. We describe DESKCHECK, a parametric static analyzer that is able to establish properties of programs that manipulate dynamically allocated memory, arrays, and integers. DESKCHECK can verify quantified invariants over mixed abstract domains, e.g., heap and numeric domains. These domains need only minor extensions to work with our domain combination framework.

The technique used for managing the communication between domains is reminiscent of the Nelson-Oppen technique for combining decision procedures, in that the two domains share a common predicate language to exchange shared facts. However, whereas the Nelson-Oppen technique is limited to a common predicate language of shared equalities, the technique described in this paper uses a common predicate language in which shared facts can be quantified predicates expressed in first-order logic with transitive closure.

We explain how we used DESKCHECK to establish memory safety of the `thttpd` web server's cache data structure, which uses linked lists, a hash table, and reference counting in a single composite data structure. Our work addresses some of the most complex data-structure invariants considered in the shape-analysis literature.

1 Introduction

Many programs use data structures for which a proof of correctness requires a combination of heap and numeric reasoning. DESKCHECK, the tool described in this paper, is targeted at such programs. For example, consider a program that uses an array, *table*, whose entries point to heap-allocated objects. Each object has an *index* field. We want to check that if $table[k] = obj$, then $obj.index = k$. In verifying the correctness of the `thttpd` web server [22], this invariant is required

⋆ Supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grant N00014-{09-1-0510}, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.

⋆⋆ Supported, in part, by grants NSF CNS-050955 and NSF CCF-0430378 with additional support from DARPA.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 71–99, 2010.
© Springer-Verlag Berlin Heidelberg 2010

even to prove memory safety. Formally, we write the following (ignoring array bounds for now):

$$\forall k{:}\mathbb{Z}.\ \forall o{:}\mathbb{H}.\ table[k] = o \Rightarrow (o.index = k \vee o = null) \tag{1}$$

We call this invariant Inv1. It quantifies over both heap objects and integers. Such quantified invariants over mixed domains are beyond the power of most existing static analyzers, which typically infer either heap invariants or integer invariants, but not both.

Our approach is to combine existing abstract domains into a single abstract interpreter that infers mixed invariants. In this paper, we discuss examples using a particular heap domain (canonical abstraction) and a particular numeric domain (difference-bound matrices). However, the approach supports a wide variety of domain combinations, including combinations of two numeric domains, and a combination of the separation-logic shape domain [9] and polyhedra.

Our goal is for the combined domain to be more than the sum of its parts: to be able to infer facts that neither domain could infer alone. As in previous research on combining domains, communication between the two domains is the crucial ingredient. The combined domain of Gulwani and Tiwari [15], based on the Nelson-Oppen technique for combining decision procedures [20], shares equalities between domains. Our technique also uses a common predicate language to share facts; however, in our approach shared facts can be predicates from first-order logic with transitive closure.

Approach. We assume that each domain being combined reasons about a distinct collection of abstract "individuals" (heap objects, or integers, say). Every domain is responsible for grouping its individuals into sets, called *classes*. A heap domain might create a class of all objects belonging to a linked list, while an integer domain may have a class of numbers between 3 and 10.

Additionally, each domain D exposes a set of n-ary predicates to other domains. Every predicate has a definition, such as "$\mathsf{R}(o_1, o_2)$ holds if object o_1 reaches o_2 via *next* edges." Only the defining domain understands the meaning of its predicates. However, quantified atomic facts are shared between domains: a heap domain D might share with another domain the fact that $(\forall o_1 \in C_1, o_2 \in C_2.\ \mathsf{R}(o_1, o_2))$, where C_1 and C_2 are classes of list nodes. Other domains can define their own predicates in terms of R. They must depend on shared information from D to know where R holds because they are otherwise ignorant of R's semantics.

Chains of dependencies can exist between predicates in different domains. A predicate P2 in domain D' can refer to a predicate P1 in D. Then a predicate P3 in D can refer to P2 in D'. The only restriction is that dependencies be acyclic. As transfer functions execute, atomic facts about predicates propagate between domains along the dependency edges. This flexibility enables our framework to reason precisely about mixed heap and numeric invariants.

A Challenging Verification Problem. We have applied DESKCHECK to the cache module of the `thttpd` web server [22]. We chose this data structure because it

relies on several invariants that require combined numeric and heap reasoning. We believe this data structure is representative of many that appear in systems code, where arrays, lists, and trees are all used in a single composite data structure, sometimes with reference counting used to manage deallocation. Along with DESKCHECK, our model of `thttpd`'s cache is available online for review [18].

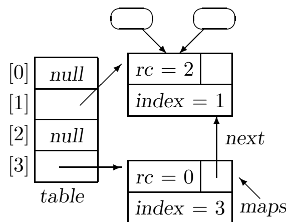


Fig. 1. `thttpd`'s cache data structure

The `thttpd` cache maps files on disk to their contents in memory. Fig. 1 displays an example of the structure. It is a composite between a hash table and a linked list. The linked list of cache entries starts at the *maps* variable and continues through *next* pointers. These same cache entries are also pointed to by elements of the *table* array. The *rc* field records the number of incoming pointers from external objects (i.e., not counting pointers from the *maps* list nor from *table*), represented by rounded rectangles. The reference count is allowed to be zero.

Fig. 2 shows excerpts of the code to add an entry to the cache. Besides the data structures already discussed, the variable *free_maps* is used to track unused cache entries (to avoid calling `malloc` and `free`). Our goal is to verify that this code, as well as the related code for releasing and freeing cache entries, is memory-safe. One obvious data-structure invariant is that *maps* and *free_maps* should point to acyclic singly linked lists of cache entries. However, there are two other invariants that are more complex but required for memory safety.

Inv1 (from Eqn. (1))**:** When a cache entry e is freed, `thttpd` nulls out its hash table entry via `table[e.index] = null` (this code is not shown in Fig. 2). If the wrong element were overwritten, then a pointer to the freed entry would remain in *table*, later leading to a segfault when accessed. Inv1 guarantees that if $table[i] = e$, where e is the element being freed, then $e.index = i$, so the correct entry will be set to null.

Inv2: This invariant relates to reference counting. The two main entry points to the cache module are called `map` and `unmap`. The `map` call creates a cache entry if it does not already exist and returns it to the caller. The caller can use the entry until it calls `unmap`. The cache keeps a reference count of the number of outstanding uses of each entry; when the count reaches zero, it is legal (although not necessary) to free the entry. Outstanding references are shown as rounded

```
1  Map * map(...)
2  { /* Expand hash table if needed */
3    check_hash_size();
4    m = find_hash(...);
5    if (m != (Map*)0) {
6      /* Found an entry */
7      ++m->refcount;
8      ...
9      return m;
10   }
11   /* Find a free Map entry
12      or make a new one. */
13   if (free_maps != (Map*)0) {
14     m = free_maps;
15     free_maps = m->next;
16   } else {
17     m = (Map*)malloc(sizeof(Map));
18   }
19   m->refcount = 1;
20   ...
21   /* Add m to hashtable */
22   if (add_hash(m) < 0) {
23     /* error handling code */
24   }
25   /* Put m on active list. */
26   m->next = maps;
27   maps = m;
28   ...
29   return m;
30 }
31 static int add_hash(Map* m)
32 { ...
33   int i = hash(m);
34   table[i] = m;
35   m->index = i;
36   ...
37 }
```

Fig. 2. Excerpts of the `thttpd` `map` and `add_hash` functions

rectangles in Fig. 1. The cache must maintain the invariant that the number of outstanding references is equal to the value of an entry's reference count (*rc*) field—otherwise an entry could be freed while still in use. We can write this invariant formally as follows. Assuming that cache entries are stored in the *entry* field of the caller's objects (the ones shown by rounded rectangles), we wish to ensure that the number of *entry* pointers to a given object is equal to its *rc* field.

$$\mathsf{Inv2} \stackrel{\text{def}}{=} \forall o{:}\mathbb{H}.\ o.rc = |\{p{:}\mathbb{H} \mid p.entry = o\}| \tag{2}$$

Verification. We give an example of how $\mathsf{Inv1}$ is verified. §4.3 has a more detailed presentation of this example. The program locations of interest are lines 34 and 35 of Fig. 2, where the hash table is updated. Recall that $\mathsf{Inv1}$ requires that if $table[k] = e$ then $e.index = k$. After line 34, $\mathsf{Inv1}$ is broken, although only "locally" (i.e., at a single index position of *table*). As a first step, we parametrize $\mathsf{Inv1}$ by dropping the quantifier on k, allowing us to distinguish between index positions at which $\mathsf{Inv1}$ is broken and those where it continues to hold.

$$\mathsf{Inv1}(k{:}\mathbb{Z}) \stackrel{\text{def}}{=} \forall o{:}\mathbb{H}.\ table[k] = o \Rightarrow (o.index = k \vee o = null)$$

After line 34 we know that $\mathsf{Inv1}(x)$ holds for all $x \neq i$. Line 35 restores $\mathsf{Inv1}(i)$.

Neither domain fully understands the defining formula of $\mathsf{Inv1}$: as we will see, the variable *table* is understood only by the heap domain whereas the field *index* is understood only by the integer domain. Consequently, we factor out the

integer portion of Inv1 into a separate predicate, as follows.

$$\mathsf{Inv1}(k{:}\mathbb{Z}) \stackrel{\text{def}}{=} \forall obj{:}\mathbb{H}.\ table[k] = o \Rightarrow (\mathsf{HasIdx}(o, k) \vee o = null)$$

$$\mathsf{HasIdx}(o{:}\mathbb{H}, k{:}\mathbb{Z}) \stackrel{\text{def}}{=} o.index = k$$

Now Inv1 is understood by the heap domain and HasIdx is understood by the integer domain.

Deskcheck splits the analysis effort between the heap domain and the numeric domain. Line 34 is initially processed by the heap domain because it assigns to a pointer location. However, the heap domain knows nothing about i, an integer. Before executing the assignment, the integer domain is asked to find an integer class containing i. Call this class N_i. Assume that all other integers are grouped into a class $N_{\neq i}$. Then the heap domain essentially treats the assignment on line 34 as $table[N_i] := m$. Since the predicate $\mathsf{HasIdx}(m, i)$ is false at this point, the assignment causes Inv1 to be falsified at N_i. Given information from the integer domain that N_i and $N_{\neq i}$ are disjoint, the heap domain can recognize that remains true at $N_{\neq i}$.

Line 35 is handled by the integer domain because the value being assigned is an integer. The heap domain is first asked to convert m to a class, H_m, so that the integer domain knows where the assignment takes place. After performing the assignment as usual, the integer domain informs the heap domain that $(\forall o \in H_m, n \in N_i.\ \mathsf{HasIdx}(o, n))$ has become true. The heap domain then recognizes that Inv1 becomes true at N_i, restoring the invariant.

Limitations. It is important to understand the limitations of our work. The most important limitation is that *shared predicates, like Inv1 and HasIdx, must be provided by the user of the analysis.* Without shared predicates, our combined domain is no more (or less) precise than the work of Gulwani et al. [14]. The predicates that we supply in our examples tend to follow directly from the properties we want to prove, but supplying their definitions is still an obligation left to the Deskcheck user. Another limitation, which applies to our implementation, is that the domains we are combining sometimes require annotations to the code being analyzed. These annotations do not affect soundness, but they may affect precision and efficiency. We describe both the predicates and the annotations we use for the `thttpd` web server in §5.

Two more limitations affect our implementation. First, it handles calls to functions via inlining. Besides not scaling to larger codebases, inlining cannot handle recursive functions. The use of inlining is not fundamental to our technique, but we have not yet developed a more effective method of analyzing procedures. We emphasize, though, that *we do not require any loop invariants or procedure pre-conditions or post-conditions from the user.* All invariants are inferred by abstract interpretation. We seed the analysis with an initially empty heap.

The final limitation is that our tool requires the user to manually translate C code to a special analysis language similar to BoogiePL [7]. This step could easily be automated, but we have not had time to do it.

Contributions. The contributions of our work can be summarized as follows: **(1)** We present a method to infer quantified invariants over mixed domains while using separate implementations of the different domains. **(2)** We describe an instantiation of Deskcheck based on canonical abstraction for heap properties and difference constraints for numeric properties. We explain how this analyzer is able to establish memory-safety properties of the `thttpd` cache. The system is publicly available online [18]. **(3)** Along with the work of Berdine et al. [2], our work addresses the most complex data-structure invariants considered in the shape-analysis literature. The problems addressed in the two papers are complementary: Berdine et al. handle complex *structural* invariants for nests of linked structures (such as "cyclic doubly linked lists of acyclic singly linked lists"), whereas our work handles complex *mixed-domain* invariants for data structures with both linkage and numeric constraints, such as the structure depicted in Fig. 1.

Organization. §2 summarizes the modeling language and the domain-communication mechanism on which Deskcheck relies. §4 describes how Deskcheck infers mixed numeric and heap properties. §5 presents experimental results. §6 discusses related work.

2 Deskcheck Architecture

2.1 Modeling of Programs

Programs are input to Deskcheck in an imperative language similar to BoogiePL [7]. We briefly describe the syntax and semantics, because this language is used in all this paper's examples. The syntax is Pascal-like. An example program is given in Fig. 3. This program checks that each entry in a linked list has a data field of zero; this field is then set to one.

Line 1 declares a type `T` of list nodes. Lines 3–5 define a set of *uninterpreted functions*. Our language uses uninterpreted functions to model variables, fields, and arrays uniformly. The *next* function models a field: it maps a list node to another list node, so its *signature* is $\mathtt{T} \rightarrow \mathtt{T}$. The *data* function models an integer field of list nodes. And *head* models a list variable; it is a nullary function. Note that an array `a` of type `T` would be written as `a[int]:T`. At line 8, *cur* is a procedure-local nullary uninterpreted function (another `T` variable).

The semantics of our programs is similar to the semantics of a many-sorted logic. Each type is a sort, and the type `int` also forms a sort. For each sort there is an infinite, fixed universe of individuals. (We model allocation and deallocation with a free list.) A concrete program state maps uninterpreted function names to mathematical functions having the correct signature. For example, if $U_{\mathtt{T}}$ is the universe of `T`-individuals, then the semantics of the *data* field is given by some function drawn from $U_{\mathtt{T}} \rightarrow \mathbb{Z}$.

```
1  type T;
2
3  global next[T]:T; global data[T]:int; global head:T;
4
5  procedure iter()
6    cur:T;
7  { cur := head;
8    while (cur != null) {
9      assert(data[cur] = 0);
10     data[cur] := 1;
11     cur := next[cur];
12   }
13 }
```

Fig. 3. A program for traversing a linked list

2.2 Base Domains

DESKCHECK combines the power of several abstract domains into a single combined domain. In our experiments, we used a combination of canonical abstraction for heap reasoning and difference-bound matrices for numeric reasoning. However, combinations using separation logic or polyhedra are theoretically possible.

Canonical abstraction [24] partitions heap objects into disjoint sets based on the properties they do or do not satisfy. For example, canonical abstraction might group together all objects reachable from a variable x but not reachable from y. When two objects are grouped together, only their common properties are preserved by the analysis. A canonical abstraction with many groups preserves more distinctions between objects but is more expensive. Using fewer groups is faster but less precise.

Canonical abstraction is a natural fit for DESKCHECK because it already relies on predicates. Each canonical name corresponds fairly directly to a class in the DESKCHECK setting. DESKCHECK allows each domain to decide how objects are to be partitioned into classes: in canonical abstraction we use predicates to decide. We use a variant of canonical abstraction in which a summary node summarizes 0 or more individuals [1] (rather than 1 or more as in most other systems).

Our numeric domain is the familiar domain of difference-bound matrices. It tracks constraints of the form $t_1 - t_2 \leq c$, where t_1 and t_2 are uninterpreted function terms such as $f[x]$. We use a summarizing numeric domain [12], which is capable of reasoning about function terms as dimensions in a sound way.

The user is allowed to define numeric predicates. These predicates are defined using a simple quantifier-free language permitting atomic numerical facts, conjunction, and disjunction. A typical predicate might be $\mathsf{Bounded}(n) := n \geq$

$0 \wedge n < 10$. Similar to canonical abstraction, we use these numeric predicates to partition the set of integers into disjoint classes. These integer classes permit array reasoning, as explained later in §4.2.

2.3 Combining Domains

In the DESKCHECK architecture, work is partitioned between n domains. Typically $n = 2$, although all of our work extends to an arbitrary number of base domains. Besides the usual operations like join and assignment, these domains must be equipped to share quantified atomic facts and class information.

Each domain is responsible for some of the sorts defined above. In our implementation, the numeric domain handles `int` and the heap domain handles all other types. An uninterpreted function is associated with an abstract domain according to the type of its range. In Fig. 3, *next* and *head* are handled by the heap domain and *data* by the numeric domain. Assignments statements to uninterpreted functions are initially handled by the domain with which they are associated.

Predicates are also associated with a given domain. Each domain has its own language in which its predicates are defined. Our heap domain supports universal and existential quantification and transitive closure over heap functions. Our numeric domain supports difference constraints over numeric functions along with cardinality reasoning. A predicate associated with one domain may refer to a predicate defined in another domain, although cyclic references are forbidden. The user is responsible for defining all predicates. The precision of an analysis depends on a good choice of predicates; however, soundness is guaranteed regardless of the choice of predicates.

Classes. A class, as previously mentioned, represents a set of individuals of a given sort (integers, heap objects of some type, etc.). A class can be a singleton, having one element, or a summary class, having an arbitrary number of elements (including zero). Summary classes are written in bold, as in $\boldsymbol{N_{\neq i}}$, to distinguish them.

The grouping of individuals into classes may be flow-sensitive—we do not assume that the classes are known prior to the analysis. At any time a domain is allowed to change this grouping, in a process called *repartitioning*. Classes of a given sort are repartitioned by the domain to which that sort is assigned. When a domain repartitions its classes, other domains are informed as described below.

Semantics. Each domain D_i can choose to represent its abstract elements however it desires. To define the semantics of a combined element $\langle E_1, E_2 \rangle$, we require each domain D_i to provide a meaning function, $\widehat{\gamma}_i(E_i)$, that gives the meaning of E_i as a logical formula. This formula may contain occurrences of uninterpreted functions that are managed by D_i as well as classes and predicates managed by any of the domains.

We will define a function $\gamma(\langle E_1, E_2 \rangle)$ that gives the semantics of a combined abstract element. Instead of evaluating to a logical formula, this function returns

a set of concrete states that satisfy the constraints of E_1 and E_2. A concrete state is an interpretation that assigns values to all the uninterpreted functions used by the program.

Naively, we could define $\gamma(\langle E_1, E_2 \rangle)$ as the set of states that satisfy formulas $\widehat{\gamma_1}(E_1)$ and $\widehat{\gamma_2}(E_2)$. However, these formulas refer to classes and predicates, which do not appear in the state. To solve the problem, we let $\gamma(\langle E_1, E_2 \rangle)$ be the set of states satisfying $\widehat{\gamma_1}(E_1)$ and $\widehat{\gamma_2}(E_2)$ for *some* interpretation of predicates and classes. We can state this formally using second-order quantification. Here, each P_i is a predicate defined by D_1 or D_2. Each C_i is a class appearing in E_1 or E_2. The number of classes, $n(E_1, E_2)$, depends on E_1 and E_2.

$$\gamma(\langle E_1, E_2 \rangle) \stackrel{\text{def}}{=} \{S : S \models \exists P_1. \cdots \exists P_m.\ \exists C_1. \cdots \exists C_{n(E_1,E_2)}.\ \widehat{\gamma_1}(E_1) \wedge \widehat{\gamma_2}(E_2)\}$$

Typically, $\widehat{\gamma_i}(E_i)$ is the conjunction of three subformulas. One subformula gives meaning to the predicates defined by D_i and another gives meaning to the classes defined by D_i. The third subformula, the only one specific to E_i, gives meaning to the constraints in E_i.

We can be more specific about the forms of these three subformulas. A subformula defining a unary predicate P that holds when its argument is positive would look as follows.

$$\forall x.\ \mathsf{P}(x) \iff x > 0$$

In our implementation of the analysis, all predicate definitions must be given by the user. Note that a predicate definition may refer to another predicate (possibly one defined by another base domain). For example, the following predicate might apply to heap objects, stating that their *data* field is positive.

$$\forall o.\ \mathsf{Q}(o) \iff \mathsf{P}(data[o])$$

A subformula that defines a class C containing the integers from 0 to n would look as follows.

$$C = \{x : 0 \le x < n\}$$

Our implementation uses canonical abstraction [24] to decide how individuals are grouped into classes. Therefore, the definition of a class will always have the following form:

$$C = \{x : \mathsf{P}(x) \wedge \mathsf{Q}(x) \wedge \neg \mathsf{R}(x) \wedge \cdots\}$$

That is, the class contains exactly those object satisfying a set of unary predicates and not satisfying another set of unary predicates. Such unary predicates are called *abstraction predicates*. The user chooses which subset of the unary predicates are abstraction predicates. In theory there can be one class for every subset of the abstraction predicates, but in practice most of these classes are empty and thus not used. Because each class is defined by the abstraction predicates it satisfies (the non-negated ones), this subset of predicates is called the class's *canonical name*.

Subformulas that give meaning to the constraints in E_i are specific to the domain D_i. For example, an integer domain would include constraints like $x - y \le$

c. A heap domain might include constraints about reachability. Both domains will often include quantified facts of the following form:

$$\forall o \in C.\ \mathsf{Q}(o)$$

A domain may quantify over a class defined by any of the domains and it may use predicates from any of the domains. The predicate that appears may optionally be negated. Facts like this may be exchanged freely between domains because they are written in a common language of predicates and classes. To distinguish the more domain-specific facts like $x - y \leq c$ from the ones exchanged between domains, we surround them in angle brackets. A fact $\langle\ \cdot\ \rangle_H$ is specific to a heap domain and $\langle\ \cdot\ \rangle_N$ is specific to a numeric domain.

3 Domain Operations

This section describes the partial order and join operation of the combined domain and also the transfer function for assignment. These operations make use of their counterparts in the base domains as well as some additional functions that we explain below.

3.1 Partial Order

We can define a very naive partial-order check for the combined domain as follows.

$$\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle \iff (E_1^A \sqsubseteq_1 E_1^B) \wedge (E_2^A \sqsubseteq_2 E_2^B)$$

Here, we have assumed that $\sqsubseteq_1$ and $\sqsubseteq_2$ are the partial orders for the base domains.

However, there are two problems with this approach. The first problem is illustrated by the following example. (Assume that class C and predicate P are defined by D_1.)

$$E_1^A = \forall x \in C.\ \mathsf{P}(x) \qquad\qquad E_1^B = \text{true}$$
$$E_2^A = \text{true} \qquad\qquad E_2^B = \forall x \in C.\ \mathsf{P}(x)$$

If we work out $\gamma(\langle E_1^A, E_2^A \rangle)$ and $\gamma(\langle E_1^B, E_2^B \rangle)$, they are identical. Thus, we should obtain $\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle$. However, the partial-order check given above does not, because it is not true that $E_2^A \sqsubseteq_2 E_2^B$.

To solve this problem, we *saturate* E_1^A and E_2^A before applying the base domains' partial orders. That is, we strengthen these elements by exchanging any facts that can be expressed in a common language. (Note that E_1^A and E_2^A are individually strengthened but $\gamma(\langle E_1^A, E_2^A \rangle)$ remains the same; saturation is a semantic reduction.) In the example, the fact $\forall x \in C.\ \mathsf{P}(x)$ is copied from E_1^A to E_2^A.

Any fact drawn from the following grammar can be shared.

$$F ::= \forall x \in C.\ F \mid \exists x \in C.\ F \mid \mathsf{P}(x, y, \ldots) \mid \neg\mathsf{P}(x, y, \ldots) \tag{3}$$

Here, C is an arbitrary class and P is an arbitrary predicate. All variables appearing in $\mathsf{P}(x, y, \ldots)$ must be bound by quantifiers.

function Saturate(E_1, E_2):
  $F := \emptyset$
  repeat:
    $F_0 := F$
    $F := F \cup \mathit{Consequences}_1(E_1) \cup \mathit{Consequences}_2(E_2)$
    $E_1 := \mathit{Assume}_1(E_1, F)$
    $E_2 := \mathit{Assume}_2(E_2, F)$
  until $F_0 = F$
  return $\langle E_1, E_2 \rangle$

Fig. 4. Implementation of combined-domain saturation

To implement sharing, each domain D_i is required to expose an Assume_i function and a $\mathit{Consequences}_i$ function. $\mathit{Consequences}_i$ takes a domain element and returns all facts of the form above that it implies. Assume_i takes a domain element E and a fact f of the form above and returns an element that approximates $E \wedge f$. The pseudocode in Fig. 4 shows how facts are propagated. They are accumulated via $\mathit{Consequences}_i$ and then passed to the domains with Assume_i. Because we require that the number of predicates and classes in any element is bounded, this process is guaranteed to terminate.

We update the naive partial-order check as follows. If $\langle E_1^{A^*}, E_2^{A^*} \rangle =$ Saturate(E_1^A, E_2^A), then

$$\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle \iff (E_1^{A^*} \sqsubseteq_1 E_1^B) \wedge (E_2^{A^*} \sqsubseteq_2 E_2^B)$$

Note that we only saturate the left-hand element; strengthening the right-hand element is sound, but it does not improve precision.

This ordering is still too imprecise. The problem is that the A and B elements may use different class names to refer to the same set of individuals. As an example, consider the following.

$$E_1^A = \forall x \in C.\ \mathsf{P}(x) \qquad E_1^B = \forall x \in C'.\ \mathsf{P}(x)$$
$$E_2^A = (C = \{x : x > 0\}) \qquad E_2^B = (C' = \{x : x > 0\})$$

It's clear that C and C' both refer to the same sets. Therefore, $\gamma(\langle E_1^A, E_2^A \rangle)$ is equal to $\gamma(\langle E_1^B, E_2^B \rangle)$; the difference in naming between C and C' is irrelevant to γ because it projects out class names using an existential quantifier. However, our naive partial-order check cannot discover the equivalence.

To solve the problem, we rename the classes appearing in $\langle E_1^A, E_2^A \rangle$ so that they match the names used in $\langle E_1^B, E_2^B \rangle$. This process is done in two steps: (1) match up the classes in the A element with those in the B element, (2) rewrite the A element's classes according to step 1. In the example above, we get the rewriting $\{C \mapsto C'\}$ in step 1, which is used to rewrite E_1^A and E_2^A as follows.

$$E_1^A = \forall x \in \mathbf{C'}.\ \mathsf{P}(x) \qquad E_1^B = \forall x \in C'.\ \mathsf{P}(x)$$
$$E_2^A = (\mathbf{C'} = \{x : x > 0\}) \qquad E_2^B = (C' = \{x : x > 0\})$$

We only rewrite the A elements because rewriting may weaken the abstract element and it is unsound to weaken the B elements in a partial order check. Our partial order is sound with respect to γ, but it may be incomplete. Its completeness depends on the completeness of the base domain operations like *MatchClasses*$_i$, and typically these operations are incomplete.

Recall that each class is managed by one domain but may still be referenced by other domains. In the matching step, each domain is responsible for matching its own classes. In our implementation, we match up classes according to their canonical names. Then the rewritings for all domains are combined and every domain element is rewritten using the combined rewriting. In the example above, D_2 defines classes C and C', so it is responsible for matching them. But both E_1^A and E_2^A are rewritten.

function $\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle$:
$\quad\langle E_1^A, E_2^A \rangle := \text{Saturate}(E_1^A, E_2^A)$

$\quad R_1 := \mathit{MatchClasses}_1(E_1^A, E_1^B)$
$\quad R_2 := \mathit{MatchClasses}_2(E_2^A, E_2^B)$

$\quad E_1^{A'} := \mathit{Repartition}_1(E_1^A, R_1 \cup R_2)$
$\quad E_2^{A'} := \mathit{Repartition}_2(E_2^A, R_1 \cup R_2)$

$\quad$**return** $(E_1^{A'} \sqsubseteq_1 E_1^B) \wedge (E_2^{A'} \sqsubseteq_2 E_2^B)$

Fig. 5. Pseudocode for combined domain's partial order

Pseudocode that defines the partial-order check for the combined domain is shown in Fig. 5. First, E^A is saturated and its classes are matched to the classes in E^B. Each domain is required to expose a *MatchClasses*$_i$ operation that matches the classes it manages. The rewritings R_1 and R_2 are combined and then E^A is rewritten via the *Repartition*$_i$ operations that each domain must also expose. Finally, we apply each base domain's partial order to obtain the final result.

3.2 Join and Widening

The join algorithm is similar to the partial-order check. We perform saturation, rewrite the class names, and then apply each base domain's join operation independently. The difference is that join is handled symmetrically: both elements are saturated and rewritten. Instead of matching the classes of E^A to the classes of E^B, we allow both inputs to be repartitioned into a new set of classes that may be more precise than either of the original sets of classes. Thus, we require domains to expose a $\mathtt{MergeClasses}_i$ operation that returns a mapping from either element's original classes to new classes.

function $\langle E_1^A, E_2^A \rangle \sqcup \langle E_1^B, E_2^B \rangle$:
$\quad\langle E_1^A, E_2^A \rangle := \text{Saturate}(E_1^A, E_2^B)$
$\quad\langle E_1^B, E_2^B \rangle := \text{Saturate}(E_1^B, E_2^B)$

$\quad\langle R_1^A, R_1^B \rangle := \mathtt{MergeClasses}_1(E_1^A, E_1^B)$
$\quad\langle R_2^A, R_2^B \rangle := \mathtt{MergeClasses}_2(E_2^A, E_2^B)$

$\quad E_1^{A'} := \mathtt{Repartition}_1(E_1^A, R_1^A \cup R_2^A)$
$\quad E_2^{A'} := \mathtt{Repartition}_2(E_2^A, R_1^A \cup R_2^A)$

$\quad E_1^{B'} := \mathtt{Repartition}_1(E_1^B, R_1^B \cup R_2^B)$
$\quad E_2^{B'} := \mathtt{Repartition}_2(E_2^B, R_1^B \cup R_2^B)$

$\quad$**return** $\langle (E_1^{A'} \sqcup_1 E_1^{B'}), (E_2^{A'} \sqcup_2 E_2^{B'}) \rangle$

Fig. 6. Pseudocode for combined domain's join algorithm

The pseudocode for join is shown in Fig. 6. First, E^A and E^B are saturated. Then $\mathtt{MergeClasses}_1$ and $\mathtt{MergeClasses}_2$ are called to generate four rewritings. The rewriting R_i^A describes how to rewrite the classes in E^A that are managed by D_i into new classes. Similarly, R_i^B describes how to rewrite the classes in E^B that are managed by D_i. Finally, E^A and E^B are rewritten and the base domains' joins are applied. When rewriting E^A, we need both R_1^A and R_2^A because classes managed by one base domain can be referenced by the other.

We must define a widening operation for the combined domain as well. The widening algorithm is very similar to the join algorithm. Recall that the purpose of widening is to act like a join while ensuring that fixed-point iteration will terminate eventually. Due to the termination requirement, we make some changes to the join algorithm.

The challenging part of widening is that some widenings that are "obviously correct" may fail to terminate. Miné [19] describes how this can occur in an integer domain. Widening typically works by throwing away facts, producing a

less precise element, to reach a fixed point more quickly. The problem occurs if we try to saturate the left-hand operand. Saturation will put back facts that we might have thrown away, thereby defeating the purpose of widening. So to ensure that a widened sequence terminates, we never saturate the left-hand operand. The code is in Fig. 7.

function $\langle E_1^A, E_2^A \rangle \nabla \langle E_1^B, E_2^B \rangle$:
  $\langle E_1^B, E_2^B \rangle := \text{Saturate}(E_1^B, E_2^B)$

  $R_1 := \mathtt{MatchClasses}_1(E_1^B, E_1^A)$
  $R_2 := \mathtt{MatchClasses}_2(E_2^B, E_2^A)$

  $E_1^{B'} := \mathtt{Repartition}_1(E_1^B, R_1 \cup R_2)$
  $E_2^{B'} := \mathtt{Repartition}_2(E_2^B, R_1 \cup R_2)$

  return $\langle (E_1^A \nabla_1 E_1^{B'}), (E_2^A \nabla_2 E_2^{B'}) \rangle$

Fig. 7. Combined domain's widening algorithm

This code is very similar to the code for the join algorithm. Besides avoiding saturation of E^A, we also avoid repartitioning E^A. Our goal is to avoid any changes to E^A that might cause the widening to fail to terminate. Because we do not repartition E^A, we use $\mathtt{MatchClasses}_i$ instead of $\mathtt{MergeClasses}_i$.

3.3 Assignment

Assignment in the combined domain must solve two problems. First, each base-domain element must be updated to account for the assignment. Second, any changes to the shared predicates and classes must be propagated between domains. We simplify the matter somewhat by declaring that an assignment operation cannot affect classes. That is, the set of individuals belonging to a class is not affected by assignments. However, a predicate that once held over the members of a class may no longer hold, and vice versa.

Base Facts. We deal with updating the base domains first, and we deal with predicates later. We require each base domain to provide an assignment transfer function to process assignments. An assignment operation has the form $f[e_1, \ldots, e_k] := e$, where f is an uninterpreted function and $e, e_1, \ldots, e_k$ are all terms made up of applications of uninterpreted functions. The assignment transfer function of domain D_i is invoked as $\mathtt{Assign}_i(E_i, f[e_1, \ldots, e_k], e)$. Each uninterpreted function is understood by only one base domain; we use the transfer function of the domain that understands f. The other domain is left unchanged.

Assume that D_1 understands f so that $\mathtt{Assign}_1$ is invoked. The problem is that any of e or $e_1, \ldots, e_k$ may use uninterpreted functions that are understood by D_2 and not by D_1. In this case, D_1 will not know the effect of the assignment. To overcome this problem, we ask D_2 to replace any "foreign" term appearing in e and $e_1, \ldots, e_k$ with a class that is guaranteed to contain the individual to which the term evaluates. Because classes have meaning to both domains, it is now possible for D_1 to process the assignment.

Replacement of foreign terms with classes must be done recursively, because function applications may contain other function applications. The process is shown in pseudocode in Fig. 8 via the $\mathrm{TranslateFull}_i$ functions. The function $\mathrm{TranslateFull}_1$ replaces any D_2 terms with classes. When it sees a D_2 function application, it translates the arguments of the function application to terms understood by D_2 and then asks D_2, via the $\mathtt{Translate}_2$ function that it must expose, to replace the entire application with a class.

As an example, consider the term $f[c]$, where f is understood by D_1 and c is understood by D_2. If we call $\mathrm{TranslateFull}_1$ on this term, then c is converted by D_2 to a class, say C, that contains the value of c. The resulting term is $f[C]$, which is understandable by D_1. If, instead, we called $\mathrm{TranslateFull}_2$ on $f[c]$, we would again convert c to a class C. Then we would ask D_1 to convert $f[C]$ to a class, say F, which must contain the value of $f[x]$ for any $x \in C$. The result is a class, say F, which is understood by D_2.

Predicates. Besides returning an updated domain element, we require that the $\mathtt{Assign}_i$ transfer function return information about how the predicates defined by D_i were affected by the assignment. As an example, suppose that the assignment sets $\mathtt{x} := 0$ and predicate P is defined as $\mathsf{P}() := \mathtt{x} \geq 0$. If the old value of $\mathtt{x}$ was negative, then the assignment causes P to go from false to true. The other domain should be informed of the change because it may contain facts about P that need to be updated.

The changes are conveyed via two sets, U and C. The set C contains predicate facts that may have changed. Its members have the form $\mathsf{P}(C_1, \ldots, C_k)$, where each C_i is a class; this means that the truth of $\mathsf{P}(x_1, \ldots, x_k)$ may have changed if $x_i \in C_i$ for all i. If some predicate fact is *not* in C, then it is safe to assume that its truth is not affected by the assignment.

The set U holds facts that are known to be true after the assignment. Its members have same form as facts returned by $\mathtt{Consequences}_i$. For example, if an assignment causes P to go from true to false for all elements of a class C_0, then C would contain $\mathsf{P}(C_0)$ and U would contain $\forall x \in C_0.\ \neg\mathsf{P}(x)$.

The $\mathtt{Assign}_i$ transfer functions are required to return U and C. However, when one predicate depends on another, $\mathtt{Assign}_i$ may not know immediately how to update it. For example, if D_1 defines the predicate $\mathsf{P}() := \mathtt{x} \geq 0$ and D_2 defines $\mathsf{Q}() := \neg\mathsf{P}()$, then $\mathtt{Assign}_1$ has no way to know that a change in $\mathtt{x}$ might affect Q, because it is unaware of the definition of Q.

We use a post-processing step to update predicates like Q. We require predicates to be *stratified*. A predicate in the jth stratum can depend only on predicates in strata $< j$. Each domain must provide a function

```
function TranslateFull_1(E_1, E_2, f[e_1, ..., e_k]):
    if f ∈ D_1:
        for i ∈ [1..k]:  e'_i := TranslateFull_1(E_1, E_2, e_i)
        return f[e'_1, ..., e'_k]
    else:
        for i ∈ [1..k]:  e'_i := TranslateFull_2(E_1, E_2, e_i)
        return Translate_2(E_2, f[e'_1, ..., e'_k])

function TranslateFull_2(E_1, E_2, f[e_1, ..., e_k]):
    defined similarly to TranslateFull_1

function Assign(⟨E_1, E_2⟩, f[e_1, ..., e_k], e):
    ⟨E_1, E_2⟩ := Saturate(E_1, E_2)

    if f ∈ D_1:
        l := TranslateFull_1(E_1, E_2, f[e_1, ..., e_k])
        r := TranslateFull_1(E_1, E_2, e)
        ⟨E'_1, U, C⟩ := Assign_1(E_1, l, r)
        E'_2 := E_2
    else:
        l := TranslateFull_2(E_1, E_2, f[e_1, ..., e_k])
        r := TranslateFull_2(E_1, E_2, e)
        ⟨E'_2, U, C⟩ := Assign_2(E_2, l, r)
        E'_1 := E_1

    j := 1
    repeat:
        ⟨E'_1, U, C⟩ = PostAssign_1(E_1, E'_1, j, U, C)
        ⟨E'_2, U, C⟩ = PostAssign_2(E_2, E'_2, j, U, C)
        j := j + 1
    until j = num_strata

    return ⟨E'_1, E'_2⟩
```

Fig. 8. Pseudocode for assignment transfer function. num_strata is the total number of shared predicates.

$\mathit{PostAssign}_i(E_i, E'_i, j, U, C)$. Here, E_i is the domain element before the assignment and E'_i is the element that accounts for updates to base facts and to predicates in strata $< j$. U and C describe how predicates in strata $< j$ are affected by the assignment. The function's job is to compute updates to predicates in the jth stratum, returning new values for E'_i, U, and C. Fig. 8 gives the full pseudocode. It assumes that variable num_strata holds the number of strata.

4 Examples

4.1 Linked Lists

We begin by explaining how we analyze the code in Fig. 3. Although analysis of linked lists using canonical abstraction is well understood [24], this section illustrates our notation. First, some predicates must be specified by the user. These are standard predicates for analyzing singly linked lists with canonical abstraction [24]. The definition formulas use two forms of quantification: `tc` for irreflexive transitive closure and `ex` for existential quantification. All of these predicates are defined in the heap domain.

```
1 predicate NextTC(n1:T, n2:T) := tc(n1, n2) next;
2 predicate HeadReaches(n:T) := head = n || NextTC(head, n);
3 predicate CurReaches(n:T) := cur = n || NextTC(cur, n);
4 predicate SharedViaHead(n:T) := ex(n1:T) head = n && next[n1] = n;
5 predicate SharedViaNext(n:T) :=
6   ex(n1:T, n2:T) next[n1] = n && next[n2] = n && n1 != n2;
```

The predicate in line 1 holds between two list nodes if the second is reachable from the first via *next* pointers. The `Reaches` predicates hold when a list node is reachable from *head*/*cur*. The `Shared` predicates hold when a node has two incoming pointers, either from *head* or from another node's *next* field; they are usually false. These five predicates can constrain a structure to be an acyclic singly linked list.

On entry to the `iter` procedure in Fig. 3, we assume that *head* points to an acyclic singly linked list whose *data* fields are all zero. We abstract all the linked-list nodes into a summary heap class $\boldsymbol{L}$.

We describe the classes and shared predicates of the initial analysis state graphically as follows. Nodes represent classes and predicates are attached to these nodes.

This diagram means that there is a single class, $\boldsymbol{L}$, whose members satisfy the HeadReaches predicate and do not satisfy the CurReaches, SharedViaHead, or SharedViaNext predicates. The double circle means the node represents a summary class. We could write this state more explicitly as follows.

$$\forall x \in \boldsymbol{L}.\ \mathsf{HeadReaches}(x) \wedge \neg\mathsf{CurReaches}(x) \\ \wedge \neg\mathsf{SharedViaHead}(x) \wedge \neg\mathsf{SharedViaNext}(x)$$

This state exactly characterizes the family of acyclic singly linked lists. Predicate HeadReaches ensures that there are no unreachable garbage nodes abstracted by $\boldsymbol{L}$, and the two sharing predicates exclude the possibility of cycles. Note that no elements are reachable from *cur* because *cur* is assumed to be invalid on entry to `iter`.

In addition to these shared predicate facts, each domain also records its own private facts. In this case, we assume that the numeric domain records that the data field of every list element is zero: $\langle\ \forall x \in \boldsymbol{L}.\ \mathit{data}[x] = 0\ \rangle_N$. The remainder of the analysis is a straightforward application of canonical abstraction.

4.2 Arrays

In this section, we consider a loop that initializes to null an array of pointers (Fig. 9). The example demonstrates how we abstract arrays. A similar loop is used to initialize a hash table in the `thttpd` web server that we verify in §5.

```
1  type T;
2  global table[int]:T;
3
4  procedure init(n:int)
5    i:int;
6  { i := 0;
7    while (i < n) {
8      table[i] := null;
9      i := i+1;
10   }
11 }
```

Fig. 9. Initialize an array

Most of this code is analyzed straightforwardly by the integer domain. It easily infers the loop invariant that $0 \leq i < n$. Only the update to *table* is interesting.

Just as the heap domain partitions heap nodes into classes, the integer domain partitions integers into classes. We define predicates to help it determine a good partitioning.

```
1  predicate Lt(x:int) = 0 <= x && x < i;
2  predicate Eq(x:int) = x = i;
3  predicate Gt(x:int) = i < x && x < n;
```

With these predicates, we obtain four integer classes via canonical abstraction, $\boldsymbol{I_{lt}}$, I_i, $\boldsymbol{I_{gt}}$, and $\boldsymbol{X}$. The first three classes contain elements satisfying the three predicates above, respectively. The last class contains all other integers (those that are negative or $\geq n$). Given these classes, we infer the following loop invariant.

$\boldsymbol{I_{lt}}$	I_i	$\boldsymbol{I_{gt}}$	
◎	○	◎	$\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$
Lt	Eq	Gt	

The fact on the right is a private heap-domain fact but it can still refer to the integer class $\boldsymbol{I_{lt}}$. The ability of one domain to refer to another domain's classes is what enables mixed quantification in our system.

Using abstract interpretation, our analysis makes several passes over the loop before it infers this invariant. We write P_n to denote the state resulting from analyzing the nth iteration of the loop. In state P_0, $i = 0$ and so $\boldsymbol{I_{lt}}$ is empty. The fact $\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$ is vacuously true here, but our analysis does not infer facts about empty classes, so it is not included in P_0. However, it is *implied* by P_0 because $\boldsymbol{I_{lt}}$ is empty.

In state P_1, where $i = 1$, $\boldsymbol{I_{lt}}$ is non-empty and $\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$ is inferred from the assignment. To obtain a loop invariant, we join P_0 and P_1. Our join algorithm recognizes that the fact $\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$, which is present in P_1, is implied by P_0 (because $\boldsymbol{I_{lt}}$ is empty there) and so it includes this fact in the join result.

The assignment to *table* on line 8 of Fig. 9 proceeds as follows. Because the function *table* is heap-defined while i is defined in the numeric domain, the combined domain asks the numeric domain to "translate" i into a class. Ideally, the translation should generate the smallest possible class containing the value of i. In this case, the numeric domain can return the singleton class I_i, because it knows that I_i satisfies the Eq predicate. Then the heap domain can add $\langle\ \forall x \in I_i.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$ to the analysis state.

The increment to i re-arranges the class structure (although this happens outside the assignment transfer function, which requires classes to remain constant). The numeric domain materializes a new class for $i + 1$, which becomes I_i and merges the existing I_i with $\boldsymbol{I_{lt}}$. The resulting domain element implies the loop invariant.

After the loop exits, the loop invariant implies that *table* is null at all indexes in $\boldsymbol{I_{lt}}$, which now includes all valid array indexes.

4.3 Numeric Predicates

We now show how Inv1 (Eqn. (1)) is established in `thttpd`. The code contains the following variable definitions and predicates.

```
1 global table[int]:T, index[T]:int, size:int;
2 predicate HasIdx(e:T, x:int) := index[e] = x;
3 predicate Inv1(x:int) := all(e:T) table[x]=e => HasIdx(e, x) || e=null;
```

The intent is that $\mathit{table}[k] = e$ should imply $\mathit{index}[e] = k$. Variable *size* is the size of the *table* array. Note that HasIdx is defined in the numeric domain because it references *index*, while Inv1 is defined in the heap domain.

The procedures of interest to us are those that add and remove elements from the table. Our goal will be to prove that `add` preserves Inv1 and that `remove`, assuming Inv1 holds initially, does not leave any dangling pointers.

```
1 procedure add(i:int)
2   o:T;
```

```
3  { o := new T;
4    table[i] := o;
5    index[o] := i;
6  }
7  procedure remove(o:T)
8    i:int;
9  { i := index[o];
10   table[i] := null;
11   delete o;
12 }
```

Addition. Besides the predicates above, we create numeric predicates to partition the integers into five classes: $\boldsymbol{I_{lt}}$, I_i, $\boldsymbol{I_{gt}}$. Respectively, these are the integers between 0 and $i-1$, equal to i, greater than i but less than *size*. As before, class $\boldsymbol{X}$ holds the out-of-bounds integers.

Assume that upon entering the add procedure, we infer the following invariant (recall that we treat all functions via inlining).

$\boldsymbol{I_{lt}}$	I_i	$\boldsymbol{I_{gt}}$	$\boldsymbol{E}$	
◎	○	◎	◎	$\langle\ \forall x \in I_i.\ table[x] = null\ \rangle_H$
Inv1	Inv1	Inv1		

All existing T objects are grouped into the class $\boldsymbol{E}$. *table* is unconstrained at $\boldsymbol{I_{lt}}$ and $\boldsymbol{I_{gt}}$and we do not have any information about the HasIdx predicate.

Initially, Inv1 holds at I_i because *table* is null there. When *table* is updated in line 4, Inv1 is potentially broken because $index[o]$ may not be i. The assignment on line 5 correctly sets $index[o]$, restoring Inv1 at I_i.

The object allocated at line 3 is placed in a fresh class E'. We do not have information about HasIdx for this new class. When line 4 sets $table[i] := obj$, the assignment is initially handled by the heap domain because *table* is a heap function. In order for Inv1 to continue to hold after line 4, we would need to know that $\forall x \in E'.\ \forall y \in I_i.\ \mathsf{HasIdx}(x, y)$. But this fact does not hold because E' is a new object whose *index* field is undefined.

Inv1 is restored in line 5. The assignment is handled by the numeric domain. Besides the private fact that $\langle\ \forall x \in E'.\ index[x] = i\ \rangle_N$, it recognizes that $\forall x \in E'.\ \forall y \in I_i.\ \mathsf{HasIdx}(x, y)$. This information is shared with the heap domain in the $\mathtt{PostAssign}_i$ phase of the assignment transfer function. The heap domain then recognizes that Inv1 has been restored at I_i. Thus, procedure add preserves Inv1.

Removal. We use the same numeric abstraction used for procedure add. On entry we assume that the object that o points to is contained in a singleton class E'. All other T objects are in a class $\boldsymbol{E}$. All *table* entries are either *null* or members of $\boldsymbol{E}$ or E'. The verification challenge is to prove that $\langle\ \forall x \in (\boldsymbol{I_{lt}} \cup \boldsymbol{I_{gt}}).\ \forall y \in E'.\ table[x] \neq y\ \rangle_H$. Without this fact, after E' is deleted, we might have pointers from *table* to freed memory. These pointers might later be accessed, leading to a segfault.

Luckily, Inv1 implies the necessary disequality, as follows. We start by analyzing line 9. The integer domain handles this assignment and shares the fact that $\forall x \in E'.\ \forall y \in I_i.\ \mathsf{HasIdx}(x, y)$ holds afterwards. Importantly, because the integer domain knows that i is not in either $\boldsymbol{I_{lt}}$ or $\boldsymbol{I_{gt}}$, it also propagates $\forall x \in E'.\ \forall z \in (\boldsymbol{I_{lt}} \cup \boldsymbol{I_{gt}}).\ \neg\mathsf{HasIdx}(x, z)$. We assume as a precondition to `remove` that Inv1 holds of $\boldsymbol{I_{lt}}$, I_i, and $\boldsymbol{I_{gt}}$. The contrapositives of the implications in these Inv1 facts, together with the negated HasIdx facts, imply that $\langle\ \forall x \in (\boldsymbol{I_{lt}} \cup \boldsymbol{I_{gt}}).\ \forall y \in E'.\ table[x] \neq y\ \rangle_H$.

The assignment on line 10 is straightforward to handle in the heap domain. It recognizes that $\langle\ \forall x \in I_i.\ table[x] = null\ \rangle_H$ while preserving Inv1 at I_i(because the definition of Inv1 has a special case for null). Finally, line 11 deletes E', Because the heap domain knows that $\langle\ \forall x \in (\boldsymbol{I_{lt}} \cup I_i \cup \boldsymbol{I_{gt}}).\ \forall y \in E'.\ table[x] \neq y\ \rangle_H$, there can be no dangling pointers.

4.4 Reference Counting

In this final example, we demonstrate the analysis of the most complex feature of `thttpd`'s cache: reference counting. To analyze reference counting we have augmented the integer domain in two ways.

The first augmentation allows the numeric domain to make statements about the cardinality of a class. For each class $\boldsymbol{C}$ we introduce a numeric dimension $\#\boldsymbol{C}$, called a *cardinality variable*. Thus, we can make statements like $\langle\ \#\boldsymbol{C} \leq n{+}1\ \rangle_N$. This augmentation was described by Gulwani et al. [14]. Usually, information about the cardinality of a class is accumulated as the class grows. The typical class starts as a singleton, so we infer that $\#C = 1$. As it is repeatedly merged with other singleton classes, its cardinality increments by one. Often we can derive relationships between the cardinality of a class and loop-iteration variables as a data structure is constructed.

Besides cardinality variables, we also introduce *cardinality functions*. These functions are private to the numeric domain. We give an example below in the context of reference counting.

```
1  type T, Container;
2  global rc[T]:int, contains[Container]:T;
3
4  predicate Contains(c:Container, o:T) := contains[c] = o;
5  function RealRC(o:T) := card(c:Container) Contains(c, o); // see below
6  predicate Inv2(o:T) := rc[o] = RealRC[o];
```

There are two types here: `Container` objects hold references to `T` objects. Each `Container` object has a *contains* field to some `T` object. Each `T` object records the number of incoming *contains* edges in its *rc* field.

The heap predicate Contains merely exposes *contains* to the numeric domain. The cardinality function *RealRC* is private to the numeric domain. $RealRC[e]$ equals the number of incoming *contains* edges to e. It equals the cardinality of the set $\{c : \mathtt{Container} \mid \mathsf{Contains}(c, e)\}$. The Inv2 predicate holds if $rc[e]$ equals this value.

Our goal is to analyze the functions that increment and decrement an object's reference count. We check for memory safety.

```
1  procedure incref(c:Container, o:T)
2  { assert(contains[c]=null);
3    rc[o]:=rc[o]+1;
4    contains[c]:=o;
5  }
6
7  procedure decref(c:Container)
8    o:T;
9  { o := contains[c];
10   contains[c]:=null;
11   rc[o]:=rc[o]-1;
12   if (rc[o]=0)
13     delete o;
14 }
```

Increment. When we start, we assume that class C' holds the object pointed to by c and E' holds the object pointed to by o. Class $\boldsymbol{E}$ holds all the other T objects and class $\boldsymbol{C}$ contains all the other Container objects. Then $contains[c]$, for any $c \in \boldsymbol{C}$, points to an object from either $\boldsymbol{E}$ or E', while $contains[c']$, for $c' \in C'$, is null. We also assume reference counts are correct, so Inv2 at $\boldsymbol{E}$ and E'. This fact implies $\langle\ \forall x \in E'.\ RealRC[x] = rc[x]\ \rangle_N$. The assignment on line 3 updates this fact to $\langle\ \forall x \in E'.\ RealRC[x] = rc[x] - 1\ \rangle_N$ and makes Inv2 false at E'.

The assignment on line 4 is initially handled by the heap domain, which recognizes that $\forall x \in C'.\ \forall y \in E'.\ \mathsf{Contains}(x, y)$ now holds. When this new fact is shared with the numeric domain, it realizes that $RealRC$ increases by 1 at E', thereby restoring Inv2 at E' as desired.

Decrement. Analysis of lines 9, 10, and 11 are similar to incref. We assume that the singleton class E' holds the object pointed to by obj. Similarly, C' holds the object pointed to by c. Other Container objects belong to the class $\boldsymbol{C}$ and other T objects belong to $\boldsymbol{E}$. Line 10 breaks Inv2 at E' and line 11 restores it.

However, lines 12 and 13 are different. After line 12, the numeric domain recognizes that $\langle\ \forall x \in E'.\ rc[x] = 0\ \rangle_N$ holds. Therefore, it knows that $\langle\ \forall x \in E'.\ RealRC[x] = 0\ \rangle_N$ holds, based on the just-restored Inv2 invariant at E'. Given the definition of $RealRC$, it is then able to infer $\forall x \in (\boldsymbol{C} \cup C').\ \forall y \in E'.\ \neg\mathsf{Contains}(x, y)$. Therefore, when obj is freed at line 13, we know that there are no pointers to it, which guarantees that there will be no accesses to this freed object in the future.

5 Experiments

Our experiments were conducted on the caching code of the thttpd web server discussed in §1. Interested readers can find our complete model of the cache,

as well as the code for DESKCHECK, online [18]. The web-server cache has four entry-points. The `map` and `unmap` procedures are described in §1. Additionally, the `cleanup` entry-point is called optionally to free cache entries whose reference counts are zero; this happens in `thttpd` only when memory is running low. Finally, a `destroy` method frees all cache entries regardless of their reference count.

This functionality corresponds to 531 lines of C code, or 387 lines of code in the modeling language described in §2.1. The translation from C was done manually. The model is shorter because it elides the system calls for opening files and reading them into memory; instead, it simply allocates a buffer to hold the data. It also omits logging code and comments.

Our goal is to check that the cache does not contain any memory errors—that is, the cache does not access freed memory or fail to free unreachable memory. We also check that all array accesses are in bounds, that unassigned memory is never accessed, and that null is never dereferenced. We found no bugs in the code.

We verify the cache in the context of a simplified client. This client keeps a linked list of ongoing HTTP connections, and each connection stores a pointer to data retrieved from the cache. In a loop, the client calls either `map`, `unmap`, or `cleanup`. When the loop terminates, it calls `destroy`. At any time, many connections may share the same data.

All procedure calls are handled via inlining. There is no need for the user to specify function preconditions or postconditions. Because our analysis is an abstract interpretation, there is no need for the user to specify loop invariants either. This difference distinguishes DESKCHECK from work based on verification conditions.

All of the invariants described in §1 appear as predicate definitions in the verification. In total, thirty predicates are defined. Fifteen of them define common but important linked-list properties, such as reachability and sharing. These are all heap predicates. Another ten predicates are simple numeric range properties to define the array abstraction that is used to check the hash table. The final five are a combination of heap and numeric predicates to check Inv1 and Inv2; they are identical to the ones appearing in §4.3 and §4.4.

Deciding which predicates to provide to the analysis was a fairly simple process. However, the entire verification process took several weeks because it was intermingled with the development and debugging of DESKCHECK itself. It is difficult to estimate the effort that would be required for future verification work in DESKCHECK.

The experiments were performed on a laptop with a 1.86 GHz Pentium M processor and 1 GB of RAM (although memory usage was trivial). Tab. 1 shows the performance of the analysis. The total at the bottom is slightly larger than the sum of the entry-point times because it includes analysis of the client code as well. We currently handle procedure calls via inlining, which increases the cost of the analysis.

Table 1. Analysis times of `thttpd` analysis

Entry-point	Analysis time
`map`	28.23 s
`unmap`	9.08 s
`cleanup`	76.81 s
`destroy`	5.80 s
Total	123.47 s

```
1  procedure mmc_map(key:int):Buffer
2    m:Map;
3    b:Buffer;
4  {
5    check_hash_size();
6
7    m := find_hash(key);
8    if (m != null) {
9      Map_refcount[m] := Map_refcount[m]+1;
10     b := Map_addr[m];
11     return b;
12   }
13
14   @enable(free_maps);
15   if (free_maps != null) {
16     m := free_maps;
17     free_maps := Map_next[m];
18     Map_next[m] := null;
19   } else {
20     m := new Map;
21     Map_next[m] := null;
22   }
23   @disable(free_maps);
24
25   Map_key[m] := key;
26   Map_refcount[m] := 1;
27   b := new Buffer;
28   Map_addr[m] := b;
29
30   add_hash(m);
31
32   Map_next[m] := maps;
33   maps := m;
34
35   return b;
36 }
```

Fig. 10. Our model of the `mmc_map` function from Fig. 2

Annotations. Currently, we require some annotations from the user. These annotations never compromise the soundness of the analysis. Their only purpose is to improve efficiency or precision. One set of annotations marks a predicate as an abstraction predicate in a certain scope. There are 5 such scopes, making for 10 lines of annotations. We also use annotations to decide when to split an integer class into multiple classes. There are 14 such annotations. It seems possible to infer these annotations with heuristics, but we have not done so yet. All of these annotations are accounted for in the line counts above, as are the predicate definitions.

To give an example of the sorts of annotations required, we present our model of the `mmc_map` function in Fig. 10. The C code for this function is in Fig. 2. Note that *all* of our models are available online [18].

Virtually all of the code in Fig. 10 is a direct translation of Fig. 2 to our modeling language. The only annotations are at lines 14 and 23. These annotations temporarily designate `free_maps` as an abstraction predicate. This means that the node pointed to by `free_maps` is distinguished from other nodes in the canonical abstraction. Outside the scope of the annotations, every node reachable from the `free_maps` linked list is represented by a summary node. Because lines 16–18 remove the head of the list, it is necessary to treat this node separately or else the analysis will be imprecise. These two annotations are typical of all the abstraction-predicate annotations.

As a side note, a previous version of our analysis required loop invariants and function preconditions and postconditions from the user. We used this version of the analysis to check only the first two entry points, `map` and `unmap`. We found the annotation burden to be excessive. These two functions, along with their callees, required 1613 lines of preconditions, postconditions, and loop invariants. Undoubtedly a more expressive language of invariants would allow for more concise specifications, but more research would be required. This heavy annotation burden motivated us to focus on inferring these annotations as we do now via joins and widening.

6 Related Work

There are several methods for implementing or approximating the reduced product [6], which is the most precise refinement of the direct product. Granger's method of *local descending iterations* [13] uses a decreasing sequence of reduction steps to approximate the reduced product. The method provides a way to refine abstract *states*; in abstract *transformers*, domain elements can only interact either before or after transformer application. The *open-product* method [5] allows domain elements to interact *during* transformer application. Reps et al. [23] present a method that can implement the reduced product, for either abstract states or transformers, provided that one has a sat-solver for a logic that can express the meanings of both kinds of domain elements.

Combining Heap and Numeric Abstractions. The idea to combine numeric and pointer analysis to establish properties of memory was pioneered by Deutsch [8]. His abstraction deals with may-aliases rather precisely, but loses almost all information when the program performs destructive memory updates.

A general method for combining numeric domains and canonical abstraction was presented by Gopan et al. [12] (and was subsequently broadened to a general domain construction for functions [16]). A general method for tracking partition sizes (along with a specific instantiation of the general method) was presented by Gulwani at al. [14]. The work of Gopan et al. and Gulwani et al. are orthogonal methods: the former addresses how to abstract values of numeric fields; the latter addresses how to infer partition sizes. The present paper was inspired by these two works and generalizes both of them in several ways. For instance, we support more kinds of partition-based abstractions than the work of Gopan et al. [12], which makes the result more general, and may allow more scalable heap abstractions.

Gulwani and Tiwari [15] give a method for combining abstract interpreters, based on the Nelson-Oppen method for combining decision procedures. Their method also creates an abstract domain that is a refinement of the reduced product. As in Nelson-Oppen, communication between domains is solely via equalities, whereas in our method communication is in terms of classes and quantified, first-order predicates.

Emmi et al. [11] handle reference counting using auxiliary functions and predicates similar to the ones discussed in §4.4. As long as only a finite number of sources and targets are updated in a single transition, they automatically generate the corresponding updates to their auxiliary functions. For abstraction, they use Skolem variables to name single, but arbitrary, objects. Their combination of techniques is specifically directed at reference counting; it supports a form of universal quantification (via Skolem variables) to track the cardinality of reference predicates. In contrast, we have a parametric framework for combining domains, as well as a specific instantiation that supports universal and existential quantification, transitive closure, and cardinality. Their analyzer supports concurrency and ours does not. Because their method is unable to reason about reachability, their method would not be able to verify our examples (or `thttpd`).

Reducing Pointer to Integer Programs. In [10,3,17], an initial transformation converts pointer-manipulating programs into integer programs to allow integer analysis to check the desired properties. These "reduction-based approaches" uses various integer analyzers on the resulting program. For proving simple properties of singly linked lists, it was shown in [3] that there is no loss of precision; however, the approach may lose precision in cases where the heap and integers interact in complicated ways. The main problem with the approach is that the proof of the integer program cannot use any quantification. Thus, while it can make statements about the size of a local linked list, it cannot make a statement about the size of every list in a hash table. In particular, Inv1 and Inv2 both lie outside the capabilities of reduction-based approaches. Our approach alternates

between the two abstractions, allows information to flow in both directions, and can use quantification in both domains. Furthermore, the framework is parametric; in particular, it can use a separation-logic domain [9] or canonical abstraction [24] (and is not restricted to domains that can represent only singly linked lists). Finally, proving soundness in our case is simpler.

Decision Procedures for Reasoning about the Heap and Arithmetic. One of the challenging problems in the area of theorem proving and decision procedures is to develop methods for reasoning about arithmetic and quantification.

Nguyen et al. [21] present a logic-based approach that involves providing an entailment procedure. The logic allows for user-defined, well-founded inductive predicates for expressing shape and size properties of data structures. Their approach can express invariants that involve other numeric properties of data structures, such as heights of trees. However, their approach is limited to separation logic, while ours is parameterized by the heap and numeric abstractions and can be used in more general contexts. In addition, their approach cannot handle quantified cardinality properties, such as the refcount property from `thttpd`:

$$\forall v\colon v.rc = |\{u : u.f = v\}|.$$

Finally, their approach does not infer invariants, which means that a heavy annotation burden is placed on the user. In contrast, our approach is based on abstract interpretation, and can thus infer invariants of loops and recursive procedures.

The logic of Zee et al. [26,25] also permits verification of invariants involving pointers and cardinality. However, as above, this technique requires user-specified loop invariants. Additionally, the logic is sufficiently expressive that user assistance is required to prove entailment (similar to the partial order in an abstract interpretation). Because the invariants that we infer are more structured, we can prove entailment automatically. However, our abstraction annotations are similar to the case-splitting information required by their analysis.

Work by Lahiri and Qadeer also uses a specialized logic coupled with the verification-conditions approach. They use a decidable logic, so their is no need for assistance in proving entailment. However, they still require manual loop invariants.

Parameterized Model Checking. For concurrent programs, Clarke et al. [4] introduce *environment abstraction*, along with model-checking techniques for formulas that support a limited form of numeric universal quantification (the variable expresses the problem size, à la parameterized verification) together with variables that are universally quantified over non-numeric individuals (which represent processes). Our methods should be applicable to broadening the mixture of numeric and non-numeric information that can be used to model check concurrent programs.

References

1. Arnold, G.: Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 204–220. Springer, Heidelberg (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
4. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
5. Cortesi, A., Charlier, B.L., Hentenryck, P.V.: Combinations of abstract domains for logic programming. SCP 38(1-3), 27–71 (2000)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
7. DeLine, R., Leino, K.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (2005)
8. Deutsch, A.: Interprocedural alias analysis for pointers: Beyond k-limiting. In: PLDI, pp. 230–241 (1994)
9. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
10. Dor, N., Rodeh, M., Sagiv, M.: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In: PLDI, pp. 155–167 (2003)
11. Emmi, M., Jhala, R., Kohler, E., Majumdar, R.: Verifying reference counting implementations. In: TACAS (2009)
12. Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 512–529. Springer, Heidelberg (2004)
13. Granger, P.: Improving the results of static analyses programs by local decreasing iteration. In: Shyamasundar, R.K. (ed.) FSTTCS 1992. LNCS, vol. 652, Springer, Heidelberg (1992)
14. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: POPL, pp. 239–251 (2009)
15. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: PLDI (2006)
16. Jeannet, B., Gopan, D., Reps, T.: A relational abstraction for functions. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 186–202. Springer, Heidelberg (2005)
17. Magill, S., Berdine, J., Clarke, E., Cook, B.: Arithmetic strengthening for shape analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 419–436. Springer, Heidelberg (2007)
18. McCloskey, B.: Deskcheck 1.0, http://www.cs.berkeley.edu/~billm/deskcheck
19. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)

20. Nelson, G., Oppen, D.: Simplification by cooperating decision procedures. TOPLAS 1(2), 245–257 (1979)
21. Nguyen, H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
22. Poskanzer, J.: thttpd - tiny/turbo/throttling http server, http://acme.com/software/thttpd/
23. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
24. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS 24(3), 217–298 (2002)
25. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: ACM Conf. Programming Language Design and Implementation, PLDI (2008)
26. Zee, K., Kuncak, V., Rinard, M.: An integrated proof language for imperative programs. In: PLDI, pp. 338–351 (2009)

From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis

Elvira Albert[1], Puri Arenas[1], Samir Genaim[1], German Puebla[2], and Diana Vanessa Ramírez Deantes[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] DLSIIS, Technical University of Madrid (UPM), Spain

Abstract. Static analysis which takes into account the value of data stored in the heap is typically considered complex and computationally intractable in practice. Thus, most static analyzers do not keep track of *object fields* (or fields for short), i.e., they are *field-insensitive*. In this paper, we propose *locality conditions* for soundly converting fields into *local variables*. This way, field-insensitive analysis over the transformed program can infer information on the original fields. Our notion of locality is *context-sensitive* and can be applied both to numeric and reference fields. We propose then a *polyvariant* transformation which actually converts object fields meeting the locality condition into variables and which is able to generate multiple versions of code when this leads to increasing the amount of fields which satisfy the locality conditions. We have implemented our analysis within a termination analyzer for Java bytecode.

1 Introduction

When data is stored in the heap, such as in object fields (numeric or references), keeping track of their value during static analysis becomes rather complex and computationally expensive. Analyses which keep track (resp. do not keep track) of object fields are referred to as *field-sensitive* (resp. *field-insensitive*). In most cases, neither of the two extremes of using a fully field-insensitive analysis or a fully field-sensitive analysis is acceptable. The former produces too imprecise results and the latter is often computationally intractable. There has been significant interest in developing techniques that result in a good balance between the accuracy of analysis and its associated computational cost. A number of heuristics exist which differ in how the value of fields is modeled. A well-known heuristics is *field-based* analysis, in which only one variable is used to model all instances of a field, regardless of the number of objects for the same class which may exist in the heap. This approach is efficient, but loses precision quickly.

Our work is inspired on a heuristic recently proposed in [3] for numeric fields. It is based on analyzing the behaviour of *program fragments* (or *scopes*) rather than the application as a whole, and modelling only those numeric fields whose behaviour is reproducible using local variables. In general, this is possible when two sufficient conditions hold within the scope: (a) the memory location where the field is stored does not change, and (b) all accesses (if any) to such memory

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 100–116, 2010.
© Springer-Verlag Berlin Heidelberg 2010

location are done through the same reference (and not through aliases). In [3], if both conditions hold, instructions involving the field access can be *replicated* by equivalent instructions using a local variable, which we refer to as *ghost* variable. This allows using a field-insensitive analysis in order to infer information on the fields by reasoning on their associated ghost variables.

Unfortunately, the techniques proposed in [3] for numeric fields are not effective to handle reference fields. Among other things, tracking reference fields by replicating instructions is problematic since it introduces undesired aliasing between fields and their ghost variables. Very briefly, to handle reference fields, the main open issues, which are contributions of this paper, are:

– *Locality condition:* the definition (and inference) of effective locality conditions for both numeric and reference fields. In contrast to [3], our locality conditions are context-sensitive and take must-aliasing context information into account. This allows us to consider as local certain field signatures which do not satisfy the locality condition otherwise.
– *Transformation:* an automatic transformation which *converts* object fields into ghost variables, based on the above locality. We propose a combination of context-sensitive locality with a *polyvariant* transformation which allows introducing multiple versions of the transformed scopes. This leads to a larger amount of field signatures which satisfy their locality condition.

Our approach is developed for object-oriented *bytecode*, i.e., code compiled for *virtual machines* such as the Java virtual machine [10] or .NET. It has been implemented in the COSTA system [4], a COSt and Termination Analyzer for Java Bytecode. Experimental evaluation has been performed on benchmarks which make extensive use of object fields and some of them use common patterns in object-oriented programming such as enumerators and iterators.

2 Motivation: Field-Sensitive Termination Analysis

Automated techniques for proving termination are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. Analysis should keep track of how the size of the data involved in loop guards changes when the loop goes through its iterations. This information is used for determining (the existence of) a *ranking function* for the loop, which is a function which strictly decreases on a well-founded domain at each iteration of the loop. This guarantees that the loop will be executed a finite number of times. For numeric data, termination analyzers rely on a *value* analysis which approximates the value of numeric variables (e.g. [7]). Some field-sensitive value analyses have been developed over the last years (see [11,3]). For heap-allocated data structures, *path-length* [15] is an abstract domain which provides a safe approximation of the length of the longest reference chain reachable from the variables of interest. This allows proving termination of loops which traverse acyclic data structures such as linked lists, trees, etc.

```
class Iter implements Iterator {
  List state;
  List aux;
  boolean hasNext() {
    return (this.state != null);
  }
  Object next() {
    List obj = this.state;
    this.state = obj.rest;
    return obj;
  }
}
class Aux {
  int f;
}
class List {
  int data;
  List rest;
}
```

```
class Test {
  static void m(Iter x, Aux y, Aux z){
    while (x.hasNext()) x.next();
    y.f--;z.f--;
  }
  static void r(Iter x, Iter y, Aux z){
    Iter w=null;
    while (z.f > 0) {
      if ( z.f > 10 ) w=x else w=y;
      m(w,z,z);
    }
  }
  static void q(Iter x, Aux y, Aux z){
    m(x,y,z);
  }
  static void s(Iter x, Iter y, Aux w, Aux z){
    q(y,w,z);
    r(x,y,z);
  }
}
```

Fig. 1. Iterator-like example

Example 1. Our motivating example is shown in Fig. 1. This is the simplest example we found to motivate all aspects of our proposal. By now, we focus on method m. In object-oriented programming, the *iterator pattern* (also *enumerator*) is a design pattern in which the elements of a *collection* are traversed systematically using a *cursor*. The cursor points to the current element to be visited and there is a method, called next, which returns the current element and advances the cursor to the next element, if any. In order to simplify the example, the method next in Fig. 1 returns (the new value of) the cursor itself and not the element stored in the node. The important point, though, is that the state field is updated at each call to next. These kind of traversal patterns pose challenges in static analysis and effective solutions are required (see [18]). The challenges are mainly related to two issues: (1) Iterators are usually implemented using an auxiliary class which stores the cursor as a field (e.g., the "state" field). Hence, field-sensitive analysis is required; (2) The cursor is updated using a method call (e.g., within the "next" method). Hence, inter-procedural analysis is required.

We aim at inferring that the while loop in method m terminates. This can be proven by showing that the path-length of the structure pointed to by the cursor (i.e., x.state) decreases at each iteration. Proving this automatically is far from trivial, since many situations have to be considered by a static analysis. For example, if the value of x is modified in the loop body, the analysis must infer that the loop might not terminate since the memory location pointed to by x.state changes (see condition (a) in Sec. 1). The path-length abstract domain, and its corresponding abstract semantics, as defined in [15] is field-insensitive in

the sense that the elements of such domain describe path-length relations among local variables only and not among reference fields. Thus, analysis results do not provide explicit information about the path-length of reference fields.

Example 2. In the loop in method m in Fig. 1, the path-length of x cannot be guaranteed to decrease when the loop goes through its iterations. This is because x might reach its maximal chain through the field aux and not state. However, the path-length of x.state decreases, which in turn can be used to prove termination of the loop. To infer such information, we need an analysis which is able to model the path-length of x.state and not that of x. Namely, we need a field-sensitive analysis based on path-length, which is one of our main goals in this paper.

The basic idea in our approach is to replace field accesses by accesses to the corresponding *ghost* variables whenever they meet the *locality condition* which will be formalized later. This will help us achieve the two challenges mentioned above: (1) make the path-length analysis field-sensitive, (2) have an inter-procedural analysis by using ghost variables as output variables in method calls.

3 A Simple Imperative Bytecode

We formalize our analysis for a simple *rule-based* imperative language [3] which is similar in nature to other representations of bytecode [17,9]. A *rule-based program* consists of a set of *procedures* and a set of classes. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to this grammar:

$$\begin{aligned}
rule &::= p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \leftarrow g, b_1, \ldots, b_n \\
g &::= true \mid exp_1 \; op \; exp_2 \mid \mathsf{type}(x, C) \\
b &::= x{:=}exp \mid x{:=}\mathsf{new}\ \mathrm{C} \mid x{:=}y.f \mid x.f{:=}y \mid q(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \\
exp &::= \mathsf{null} \mid aexp \\
aexp &::= x \mid n \mid aexp{-}aexp \mid aexp{+}aexp \mid aexp{*}aexp \mid aexp/aexp \\
op &::= \; > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{aligned}$$

where $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, \mathrm{C})$, which succeeds if the runtime class of x is exactly C. A class C is a finite set of typed field names, where the type can be integer or a class name. The key features of this language which facilitate the formalization of the analysis are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be regarded as records, and the behavior induced by dynamic dispatch in the original bytecode program is compiled into *dispatch* rules guarded by a type check and (5) rules may have *multiple output* parameters which is useful for our transformation. The translation from (Java) bytecode to the rule-based form is performed in two steps [4]. First, a

(1) $\mathit{hasNext}(\langle \mathit{this}\rangle, \langle r\rangle) \leftarrow$
  $s_0 := \mathit{this.state}$,
  $\mathit{hasNext}_1(\langle \mathit{this}, s_0\rangle, \langle r\rangle)$.
(2) $\mathit{hasNext}_1(\langle \mathit{this}, s_0\rangle, \langle r\rangle) \leftarrow$
  $\mathsf{s_0} = \mathsf{null}, r := 0$.
(3) $\mathit{hasNext}_1(\langle \mathit{this}, s_0\rangle, \langle r\rangle) \leftarrow$
  $\mathsf{s_0} \neq \mathsf{null}, r := 1$.
(4) $\mathit{next}(\langle \mathit{this}\rangle, \langle r\rangle) \leftarrow$
  $\mathit{obj} := \mathit{this.state}, s_0 := \mathit{obj.rest}$,
  $\mathit{this.state} := s_0, r := \mathit{obj}$.
(5) $m(\langle x, y, z\rangle, \langle\rangle) \leftarrow$
  $\mathit{while}(\langle x\rangle, \langle\rangle)$,
  $s_0 := y.f, s_0 := s_0 - 1, y.f := s_0$,
  $s_0 := z.f, s_0 := s_0 - 1, z.f := s_0$.
(6) $\mathit{while}(\langle x\rangle, \langle\rangle) \leftarrow$
  $\mathit{hasNext}(\langle x\rangle, \langle s_0\rangle)$,
  $m_1(\langle x, s_0\rangle, \langle\rangle)$.
(7) $m_1(\langle x, s_0\rangle, \langle\rangle) \leftarrow$
  $\mathsf{s_0} \neq \mathsf{null}$,
  $\mathit{next}(\langle x\rangle, \langle s_0\rangle)$,
  $\mathit{while}(\langle x\rangle, \langle\rangle)$.
(8) $m_1(\langle x, y, z, s_0\rangle, \langle\rangle) \leftarrow$
  $\mathsf{s_0} = \mathsf{null}$.
(9) $r(\langle x, y, z\rangle, \langle\rangle) \leftarrow$
  $w := \mathsf{null}$,
  $r_1\langle x, y, z, w\rangle, \langle\rangle)$.
(10) $r_1(\langle x, y, z, w\rangle, \langle\rangle) \leftarrow$
  $s_0 := z.f$,
  $r_2(\langle x, y, z, w, s_0\rangle, \langle\rangle)$.
(11) $r_2(\langle x, y, z, w, s_0\rangle, \langle\rangle) \leftarrow$
  $\mathbf{s_0} > \mathbf{0}, s_0 := z.f$,
  $r_3(\langle x, y, z, w, s_0\rangle, \langle\rangle)$.
(12) $r_2(\langle x, y, z, w, s_0\rangle, \langle\rangle) \leftarrow$
  $\mathbf{s_0} \leq \mathbf{0}$.
(13) $r_3(\langle x, y, z, w, s_0\rangle, \langle\rangle) \leftarrow$
  $\mathbf{s_0} > \mathbf{10}, w := x$,
  $r_4(\langle x, y, z, w\rangle, \langle\rangle)$.
(14) $r_3(\langle x, y, z, w, s_0\rangle, \langle\rangle) \leftarrow$
  $\mathbf{s_0} \leq \mathbf{10}, w := y$,
  $r_4(\langle x, y, z, w\rangle, \langle\rangle)$.
(15) $r_4(\langle x, y, z, w\rangle, \langle\rangle) \leftarrow$
  $m(\langle x, z, z\rangle, \langle\rangle)$,
  $r_1(\langle x, y, z, w\rangle, \langle\rangle)$.
(16) $q(\langle x, y, z\rangle, \langle\rangle) \leftarrow$
  $m(\langle x, y, z\rangle, \langle\rangle)$.
(17) $s(\langle x, y, z, w\rangle, \langle\rangle) \leftarrow$
  $q(\langle y, w, z\rangle, \langle\rangle)$,
  $r(\langle x, y, z\rangle, \langle\rangle)$.

Fig. 2. Intermediate representation of running example in Fig. 1

control flow graph is built. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. For simplicity, our language does not include advanced features of Java, but our implementation deals with full *sequential* Java bytecode. The execution of rule-based programs mimics standard bytecode [10]. A thorough explanation of the latter is outside the scope of this paper.

Example 3. Fig. 2 shows the rule-based representation of our running example. Procedure m corresponds to method m, which first invokes procedure *while* as defined in rules (6)–(8). Observe that loops are extracted into separate procedures. Upon return from the *while* loop, the assignment $s_0 := y.f$ pushes the value of the numeric field $y.f$ on the stack. Then, this value is decremented by one and the result is assigned back to $y.f$. When a procedure is defined by more than one rule, each rule is *guarded* by a (mutually exclusive) condition. E.g., procedure r_2 is defined by rules (11) and (12). They correspond to checking the condition of the *while* loop in method r and are guarded by the conditions $\mathbf{s_0} > \mathbf{0}$ and $\mathbf{s_0} \leq \mathbf{0}$. In the first case, the loop body is executed. In the second case execution exits the loop. Another important observation is that all rules have input and output parameters, which might be empty. The analysis is developed on the intermediate representation, hence all references to the example in the following are to this representation and not to the source code.

4 Preliminaries: Inference of Constant Access Paths

When transforming a field f into a local variable in a given code fragment, a necessary condition is that whenever f is accessed, using $x.f$, during the

execution of that fragment, the variable x must refer to the same heap location (see condition (a) in Sec. 1). This property is known as *reference constancy* (or local aliasing) [3,1]. This section summarizes the *reference constancy analysis* of [3] that we use in order to approximate when a reference variable is constant at a certain program point. Since the analysis keeps information for each program point, we first make all program points unique as follows. The k-th rule $p(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \leftarrow g, b_1^k, \ldots, b_n^k$ has $n{+}1$ program points. The first one, $(k, 1)$, after the execution of the guard g and before the execution of b_1, until $(k, n{+}1)$ after the execution of b_n. This analysis receives a code fragment S (or scope), together with an entry procedure $p(\langle\bar{x}\rangle, \langle\bar{y}\rangle)$ from which we want to start the analysis. It assumes that each reference variable x_i points to an initial memory location represented by the symbol l_i, and each integer variable has the (symbolic) value ℓ_{num} representing any integer. The result of the analysis associates each variable at each program point with an *access path*. For a procedure with n input arguments, the entry is written as $p(l_1, \ldots, l_n)$.

Definition 1 (access path). *Given a program P with an entry $p(l_1, \ldots, l_n)$, an* access path ℓ *for a variable y at program point (k, j) is a syntactic construction which can take the forms:*

- ℓ_{any}. *Variable y might point to any heap location at (k, j).*
- ℓ_{num} *(resp. ℓ_{null}). Variable y holds a numeric value (resp.* null*) at (k, j).*
- $l_i.f_1 \ldots f_h$. *Variable y always refers to the same heap location represented by $l_i.f_1 \ldots f_h$ whenever (k, j) is reached.*

we use $acc_path(y, b_j^k)$ to refer to the access path of y before instruction b_j^k.

Intuitively, the access path $l_i.f_1 \ldots f_h$ of y refers to the heap location which results from dereferencing the i-th input argument x_i using $f_1 \ldots f_h$ in the initial heap. In other words, variable y *must alias* with $x_i.f_1 \ldots f_n$ (w.r.t. to the initial heap) whenever the execution reaches (k, j).

Example 4. Consider an execution which starts from a call to procedure m in Fig. 2. During such execution, the reference x is constant. Thus, x always refers to the same memory location within method m, which, in this case, is equal to the initial value of the first argument of m. Importantly, the content of this location can change during execution. Indeed, x is constant and thus $x.state$ always refers to the same location in the heap. However, the value stored at $x.state$ (which is in turn a reference) is modified at each iteration of the while loop. In contrast, reference $x.state.rest$ is not constant in m, since *this.state* refers to different locations during execution. Reference constancy analysis is the component that automatically infers this information. More concretely, applying it to rules ① – ③ w.r.t. the entry *hasNext*(l_1), it infers that at program point $(1, 1)$ the variable *this* is constant and always refers to l_1. Applying it to ④, it infers that: at program points $(4, 1)$, $(4, 3)$ and $(4, 4)$ the variable *this* is constant and always refers to l_1; at $(4, 2)$ variable *obj* is constant and always refers to $l_1.state$; and at $(4, 3)$, variable s_0 is constant and always refers to $l_1.state.rest$.

Clearly, references are often not globally constant, but they can be constant when we look at smaller fragments. For example, when considering an execution that starts from m, i.e., $m(l_1, l_2, l_3)$, then variable *this* in *next* and *hasNext* always refers to the same heap location l_1. However, if we consider an execution that starts from s, i.e., $s(l_1, l_2, l_3, l_4)$, then variable *this* in *next* and *hasNext* might refer to different locations l_1 or l_2, depending on how we reach the corresponding program points (through r or q). As a consequence, from a global point of view, the accesses to *state* are not constant in such execution, though they are constant if we look at each sub-execution alone. Fortunately, the analysis can be applied compositionally [3] by partitioning the procedures (and therefore rules) of P into groups which we refer to as *scopes*, provided that there are no mutual calls between scopes. Therefore, the strongly connected components (SCCs) of the program are the smallest scopes we can consider. In [3], the choice of scopes directly affects the precision and an optimal strategy does not exist: sometimes enlarging one scope might improve the precision at one program point and make it worse at another program point. Instead, in this paper, due to the context-sensitive nature of the analysis, information is propagated among the scopes and the maximal precision is guaranteed when scopes are as small as possible, i.e., at the level of SCCs. In the examples, sometimes we enlarge them to simplify the presentation. Moreover, we assume that each SCC has a single entry, this is not a restriction since otherwise the analysis can be repeated for each entry.

Example 5. The rules in Fig. 2 can be divided into the following scopes: $S_{hasNext} = \{hasNext, hasNext_1\}$, $S_{next} = \{next\}$, $S_m = \{m, while, m_1\}$, $S_r = \{r, r_1, r_2, r_3, r_4\}$, $S_q = \{q\}$ and $S_s = \{s\}$. A possible reverse topological order for the scopes of Ex. 5 is $S_{hasNext}$, S_{next}, S_m, S_r, S_q and S_s. Therefore, compositional analysis starts from $S_{hasNext}$ and S_{next} as explained in Ex. 4. Then, S_m is analyzed w.r.t. the initial call $m(l_1, l_2, l_3)$, next, S_r w.r.t. $r(l_1, l_2, l_3)$ and so on. When the call from r to m is reached, the analysis uses the reference constancy inferred for m and adapts it to the current context. This way, the reference to *state* is proven to be constant, as we have justified above. As expected, the analysis cannot guarantee that the access to *rest* is constant.

In order to decide whether a field f can be considered local in a scope S, we have to inspect all possible accesses to f in any possible execution that starts from the entry of S. Note that these accesses can appear directly in S or in a scope that is reachable from it (transitive scope). We use S^* to refer to the union of S and all other scopes reachable from S, and $S(p)$ (resp. $S(b_j^k)$) to refer to the scope in which the procedure p (resp. instruction b_j^k) is defined (resp. appears). We distinguish between access for reading the field value from those that modify its value. Given a scope S and a field signature f, the set of *read* (resp. *write*) *access paths* for f in S, denoted $R(S, f)$ (resp. $W(S, f)$), is the set of access paths of all variables y used for reading (resp. modifying) a field with the signature f, i.e., $x{:=}y.f$ (resp. $y.f{:=}x$), in S^*. Note that if S has calls to other scopes, for each call $b_j^k \equiv q(\langle\bar{w}\rangle, \langle\bar{z}\rangle) \in S$ such that $S(q) \neq S$, we should adapt the read (resp. write) access paths $R(S(q), f)$ (resp. $W(S(q), f)$) to the calling context by taking into account *aliasing* information. Let us see an example.

Example 6. The read and write sets for f, *state* and *rest* w.r.t. the scopes of Ex. 5 are:

	$R(S_i, f)$	$R(S_i, state)$	$R(S_i, rest)$	$W(S_i, f)$	$W(S_i, state)$	$W(S_i, rest)$
$S_{hasNext}$	$\{\}$	$\{l_1\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$
S_{next}	$\{\}$	$\{l_1\}$	$\{l_1.state\}$	$\{\}$	$\{l_1\}$	$\{\}$
S_m	$\{l_2, l_3\}$	$\{l_1\}$	$\{\ell_{\mathsf{any}}\}$	$\{l_2, l_3\}$	$\{l_1\}$	$\{\}$
S_r	$\{l_3\}$	$\{\ell_{\mathsf{any}}\}$	$\{\}$	$\{l_3\}$	$\{\ell_{\mathsf{any}}\}$	$\{\}$
S_q	$\{l_2, l_3\}$	$\{l_1\}$	$\{\ell_{\mathsf{any}}\}$	$\{l_2, l_3\}$	$\{l_1\}$	$\{\}$
S_s	$\{l_3, l_4\}$	$\{\ell_{\mathsf{any}}\}$	$\{\ell_{\mathsf{any}}\}$	$\{l_3, l_4\}$	$\{\ell_{\mathsf{any}}\}$	$\{\}$

Note that in S_m, we have two different read access paths l_2 and l_3 to f. However, when we adapt this information to the calling context from r, they become the same due to the aliasing of the last two arguments in the call to m from r. Instead, when we adapt this information to the calling context from q, they remain as two different access paths. Finally, when computing the sets for S_s, since we have a call to q followed by one to r, we have to merge their information and assume that there are two different access paths for f that correspond to those through the third and fourth argument of s.

5 Locality Conditions for Numeric and Reference Fields

Intuitively, in order to ensure a sound monovariant transformation, a field signature can be considered *local in a scope* S if all read and write accesses to it in *all* reachable scopes (i.e., S^*) are performed through the same access path.

Example 7. According to the above intuition, the field f is not local in m since it is not guaranteed that l_2 and l_3 (i.e., the access paths for the second and third arguments) are aliased. Therefore, f is not considered as local in S_r (since $S_m \in S_r^*$) and the termination of the while loop in r cannot be proven. However, when we invoke m within the loop body of r, we have knowledge that they are actually aliased and f could be considered local in this context.

As in [3], when applying the reference constancy analysis (and computing the read and write sets), we have assumed no aliasing information about the arguments in the entry to each SCC, i.e., we *do not know* if two (or more) input variables point to the same location. Obviously, this assumption has direct consequences on proving locality, as it happens in the example above. The following definition introduces the notion of *call pattern*, which provides *must aliasing* information and which will be used to specify entry procedures.

Definition 2 (call pattern). *A call pattern ρ for a procedure p with n arguments is a partition of $\{1, \ldots, n\}$. We denote by ρ_i the set $X \in \rho$ s.t. $i \in X$.*

Intuitively, a call pattern ρ states that each set of arguments $X \in \rho$ are guaranteed to be aliased. In what follows, we denote the most general call pattern as $\rho_\top = \{\{1\}, \ldots, \{n\}\}$, since it does not have any aliasing information.

Example 8. The call pattern for m when called from r is ρ=$\{\{1\},\{2,3\}\}$, which reflects that the 2nd and 3rd arguments are aliased. The call pattern for m when called from q is $\rho_\top$ in which no two arguments are guaranteed to be aliased.

The reference constancy analysis [3] described in Sec. 4 is applied w.r.t. $\rho_\top$. In order to obtain access path information (and read and write sets) w.r.t. a given initial call pattern ρ, a straightforward approach is to re-analyze the given scope taking into account the aliasing information in ρ. Since the analysis is compositional, another approach is to reuse the read and write access paths inferred w.r.t. $\rho_\top$ and adapt them (i.e., *rename* them) to each particular call pattern ρ. This is clearly more efficient since we can analyze the scope once and reuse the results when new contexts have to be considered. In theory, re-analyzing can be more precise, but in practice reusing the results is precise enough for our needs. The next definition provides a *renaming* operation. By convention, when two arguments are aliased, we rename them to have the same name of the one with smaller index. This is captured by the use of *min*.

Definition 3 (renaming). *Given a call pattern ρ and an access path $\ell \equiv l_i.f_1 \ldots f_n$, $\rho(\ell)$ is the* renamed access path *$l_k.f_1 \ldots f_n$ where $k = min(\rho_i)$. For a set of access paths A, $\rho(A)$ is the set obtained by renaming all elements of A.*

Example 9. Renaming the set of access paths $\{l_2, l_3\}$ obtained in Ex. 7 w.r.t. ρ of Ex. 8 results in $\{l_2\}$. This is because $\rho(l_2)$=l_2 and $\rho(l_3)$=l_2, by the convention of *min* above. It corresponds to the intuition that when calling m from r, all accesses to field f are through the same memory location, as explained in Ex. 7.

Renaming is used in the context-sensitive *locality* condition to obtain the read and write sets of a given scope w.r.t. a call pattern, using the context-insensitive sets. It corresponds to the context-sensitive version of condition (b) in Sec. 1.

Definition 4 (general locality). *A field signature f is local in a scope S w.r.t. a call pattern ρ, if $\rho(R(S,f)) \cup \rho(W(S,f)) = \{\ell\}$ and $\ell \neq \ell_{\mathsf{any}}$.*

Example 10. If we consider S_m w.r.t. the call pattern $\{\{1\},\{2,3\}\}$ then f becomes local in S_m, since l_2 and l_3 refer to the same memory location and, therefore, it is local for S_r. Considering f local in S_r is essential for proving the termination of the while loop in r. This is because by tracking the value of f we infer that the loop counter decreases at each iteration. However, making f local in all contexts is not sound, as when x and y are not aliased, then each field decreases by one, and when there are aliases, it decreases by two. Hence, in order to take full advantage of context-sensitivity, we need a *polyvariant* transformation which generates two versions for procedure m (and its successors).

An important observation is that, for reference fields, it is not always a good idea to transform all fields which satisfy the general locality condition above.

Example 11. By applying Def. 4, both reference fields *state* and *rest* are local in S_{next}. Thus, it is possible to convert them into respective ghost variables v_s (for *state*) and v_r (for *rest*). Intuitively, rule ④ in Ex. 3 would be transformed into:

$$next(\langle this, v_s, v_r\rangle, \langle v_s, v_r\rangle) \leftarrow obj{:=}v_s, s_0{:=}v_r, v_s{:=}s_0, r{:=}obj.$$

For which we cannot infer that the path-length of v_s in the output is smaller than that of v_s in the input. In particular, the path-length abstraction approximates the effect of the instructions by the constraints $\{obj{=}v_s, s_0{=}v_r, v'_s{=}s_0, r{=}obj\}$. Primed variables are due to a single static assignment. The problem is that the transformation replaces the assignment $s_0{:=}obj.rest$ with $s_0{:=}v_r$. Such assignment is crucial for proving that the path-length of v_s decreases at each call to *next*. If, instead, we transform this rule w.r.t. the field *state* only:

$$next(\langle this, v_s\rangle, \langle r, v_s\rangle){\leftarrow}obj{:=}v_s, s_0{:=}obj.rest, v_s{:=}s_0, r{:=}obj.$$

and the path-length abstraction approximates the effect of the instructions by $\{obj{=}v_s, s_0{<}obj, v'_s{=}s_0, r{=}obj\}$ which implies $v'_s{<}v_s$. Therefore, termination (of the corresponding loop) can be proven relying only on the field-insensitive version of path-length. Note that, in the second constraint, when accessing a field of an acyclic data structure, the corresponding path-length decreases.

Now we introduce a locality condition which is more restrictive than that in Def. 4, called *reference locality.* This condition is interesting because it only holds for field accesses which perform heap updates, but it does not hold for other cases. Thus, it often solves the problem of too aggressive transformation, shown above. This is achieved by requiring that the field signature is both read and written in the scope. Intuitively, this heuristics is effective for tracking the references that are used as cursors for traversing the data structures and not reference fields which are part of the data structure itself.

Definition 5 (reference locality). *A field signature f is local in a scope S w.r.t. a call pattern ρ, if $\rho(R(S, f)) = \rho(W(S, f)) = \{\ell\}$ and $\ell \neq \ell_{any}$.*

While reference locality is more effective than general locality for reference fields, in the case of numeric fields, general locality is more appropriate than reference locality. For example, numeric fields are often used to bound the loop iterations. For these cases, reference locality is not sufficient, since the field is read but not updated. Since numeric and reference fields can be distinguished by their signature, we apply general locality to numeric fields and reference locality to reference fields without problems. In what follows, we use *locality* to refer to either general or reference locality, according to the corresponding field signature.

Example 12. Field *rest* is not local in S_{next}, according to Def. 5. Field *state* is local in S_{next} and S_m, but not in $S_{hasNext}$.

6 Polyvariant Transformation of Fields to Local Variables

Our transformation of object fields to local variables is performed in two steps. First, we infer *polyvariance declarations* which indicate the (multiple) versions we need to generate of each scope to achieve a larger amount of field signatures which satisfy their locality condition. Then, we carry out a (polyvariant) transformation based on the polyvariance declarations.

We first define an auxiliary operation which, given a scope S and a call pattern ρ, infers the *induced* call patterns to the external procedures.

Definition 6 (induced call pattern). *Given a call pattern ρ for a scope S, and a call $b_k^j = q(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \in S$ such that $S(q) \neq S$, the call pattern for $S(q)$ induced by b_k^j, denoted $\rho(b_k^j)$, is obtained as follows:*

(1) generate the tuple $\langle \ell_1, \ldots, \ell_n \rangle$ where $\ell_i = \rho(acc_path(b_k^j, x_i))$; and
(2) i and h belong to the same set in $\rho(b_k^j)$ if and only if $\ell_i = \ell_h \neq \ell_{any}$.

The above definition relies on function *acc_path*(a,s) defined in Def. 1.

Example 13. Consider the scope S_r and a call pattern $\rho = \{\{1\}, \{2\}, \{3\}\}$. The call pattern induced by $b_1^{15} \equiv m(\langle x, z, z\rangle, \langle\rangle)$ is computed as follows: (1) using the access path information, we compute the access paths for the arguments $\langle x, z, z\rangle$ which in this case are $\langle l_1, l_3, l_3\rangle$; (2) $\rho(b_1^{15})$ is defined such that i and j are in the same set if the access paths of the i-th and the j-th arguments are equal. Namely, we obtain $\rho(b_1^{15}) = \{\{1\}, \{2, 3\}\}$ as induced call pattern.

Now, we are interested in finding out the maximal polyvariance level which must be generated for each scope. Intuitively, starting from the entry procedure, we will traverse all reachable scopes in a top-down manner by applying the *polyvariance operator* defined below. This operator distinguishes two sets of fields:

- F_{pred} is the set of field signatures which are local for the predecessor scope;
- F_{curr} is the set of tuples which contain a field signature and its access path, which are local in the current scope and not in the predecessor one.

This distinction is required since before calling a scope, the fields in F_{curr} should be initialized to the appropriate values, and upon exit the corresponding heap locations should be modified. Those in F_{pred} do not require this initialization. Intuitively, the operator works on a tuple $\langle S, F_{pred}, \rho\rangle$ formed by a scope identifier S, a set of predecessor fields F_{pred}, and a call pattern ρ. At each iteration, a *polyvariance declaration* of the form $\langle S, F_{pred}, F_{curr}, \rho\rangle$ is generated for the current scope, where the local fields in F_{curr} for context ρ are added. The operator transitively applies to all reachable scopes from S.

Definition 7 (polyvariance). *Given a program P with an entry procedure p and call pattern ρ, the set of all versions in P is $\mathcal{V}_P = \mathcal{P}ol(\langle S(p), \emptyset, \rho\rangle)$ s.t.*

$$\begin{aligned}\mathcal{P}ol(\langle S, F_{pred}, \rho\rangle) = \{\langle S, F_{pred}, F_{curr}, \rho\rangle\} \bigcup \\ \{\mathcal{P}ol(\langle S(q), F, \rho(b_k^j)\rangle) \mid b_j^k \equiv q(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \text{ is external call in } S\}\end{aligned}$$

where F_{curr} and F are defined as follows:

- *$F_{curr} = \{\langle f, \ell\rangle \mid f$ is local in S w.r.t. ρ with an access path ℓ and $f \notin F_{pred}\}$,*
- *$F = (F_{pred} \cup \{f \mid \langle f, \ell\rangle \in F_{curr}\}) \cap \mathsf{fields}(S^*(q))$ where $\mathsf{fields}(S^*(q))$ is the set of fields that are actually accessed in $S^*(q)$.*

Since there are no mutual calls between any scopes, termination is guaranteed.

1. Given $F{=}\{f|\langle f,\ell\rangle{\in}F_{curr}\}$, we let $\bar{v}{=}\{v_f|f{\in}F_{pred}{\cup}F\}$ be a tuple of ghost variable names.
2. **Add arguments to internal calls:** each head of a rule or a call $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)$ where p is defined in S is replaced by $p{\cdot}\mathtt{i}(\langle\bar{x}\cdot\bar{v}\rangle,\langle\bar{y}\cdot\bar{v}\rangle)$.
3. **Transform field accesses:** each access $x.f$ is replaced by v_f.
4. **Handle external calls:** let $b_j^k \equiv q(\langle\bar{x}\rangle,\langle\bar{y}\rangle) \in S$ such that $S(q) \neq S$.
   - Lookup the (unique) version $\langle S(q), F_{pred} \cup F, F', \rho(b_j^k)\}\rangle$ of $S(q)$ that matches the calling context and has a unique identifier `id`.
   - Let $\bar{v}' = \{v_f|f \in F_{pred} \cup F\} \cup \{v_f|\langle f,\ell\rangle \in F'\}$.

   Then, we transform b_j^k as follows:

   (a) **Initialization**: $\forall\langle f, l_h.f_1 \ldots f_n\rangle \in F'$ we add an initialization statement (before the call) $v_f{:=}x_h.f_1 \ldots f_n.f$;

   (b) **Call**: we add the modified call $q{\cdot}\mathtt{id}(\langle\bar{x}\cdot\bar{v}'\rangle,\langle\bar{y}\cdot\bar{v}'\rangle)$

   (c) **Recovery**: $\forall\langle f, l_h.f_1 \ldots f_n\rangle \in F'$ we add a recovering statement (after the call) $x_h.f_1 \ldots f_n.f{:=}v_f$;

Fig. 3. Transformation of a Polyvariance Declaration with Identifier `i`

Example 14. The polyvariance declarations obtained by iteratively applying $\mathcal{P}ol$ starting from $\mathcal{P}ol(\langle S_s, \emptyset, \{\{1\},\{2\},\{3\},\{4\}\}\rangle)$ are:

Id	S	F_{pred}	F_{curr}	ρ
1	S_s	$\emptyset$	$\emptyset$	$\{\{1\},\{2\},\{3\},\{4\}\}$
2	S_q	$\emptyset$	$\{l_1.state\}$	$\{\{1\},\{2\},\{3\}\}$
3	S_r	$\emptyset$	$\{l_3.f\}$	$\{\{1\},\{2\},\{3\}\}$
4	S_m	$\{f\}$	$\{l_1.state\}$	$\{\{1\},\{2,3\}\}$
5	S_m	$\{state\}$	$\emptyset$	$\{\{1\},\{2\},\{3\}\}$
6	S_{next}	$\{state\}$	$\emptyset$	$\{\{1\}\}$
7	$S_{hasNext}$	$\{state\}$	$\emptyset$	$\{\{1\}\}$

Each line defines a polyvariance declaration $\langle S, F_{pred}, F_{curr}, \rho\rangle$ as in Def. 7. The first column associates to each version a unique Id that will be used when transforming the program. The only scope with more than one version is S_m, which has two, with identifiers 4 and 5. The call patterns in such versions are different and in this case they result in different fields being local.

In general, the set of polyvariance declarations obtained can include some versions which do not result in further fields being local. Before materializing the polyvariant program, it is possible to introduce a minimization phase (see, e.g., [13]) which is able to reduce the number of versions without losing opportunities for considering fields local. This can be done using well-known algorithms [8] for minimization of deterministic finite automata.

Given a program P and the set of all polyvariance declarations $\mathcal{V}_P$, we can proceed to transform the program. We assume that each version has a unique identifier (a positive integer as in the example above) which will be used in order to avoid name clashing when cloning the code. The instrumentation is done by cloning the original code of each specification in $\mathcal{V}_P$. The clone for a polyvariance declaration $\langle S, F_{pred}, F_{curr}, \rho\rangle \in \mathcal{V}_P$ with identifier `i` is done using the algorithm in Fig. 3. The four steps of the instrumentation work as follows:

(1) This step generates new unique variable names $\bar{v}$ for the local heap locations to be tracked in the scope S. Since the variable name v_f is associated to the field signature f (note that in bytecode field signatures include class and package information), we can retrieve it at any point we need it later.

(2) This step adds the identifier `i`, as well as the tuple of ghost variables (generated in the previous step) as input and output variables to all rules which belong to the scope S in order to carry their values around during execution.

(3) This step replaces the actual heap accesses (i.e., read and write field accesses) by accesses to their corresponding ghost variables. Namely, an access $x.f$ is replaced by the ghost variable v_f which corresponds to f.

(4) Finally, we transform external calls(i.e., calls to procedures in other scopes). The main point is to consider the correct version `id` for that calling context by looking at the polyvariance declarations. Then, the call to q is replaced as follows:

 4a We first need to initialize the ghost variables which are local in $S(q)$ but not in S, namely the variables in F'.

 4b We add a call to q which includes the ghost variables $\bar{v}'$.

 4c After returning from the call, we *recover* the value of the memory locations that correspond to ghost variables which are local in $S(q)$ but not in S, i.e., we put their value back in the heap.

Note that in points 4a and 4c, it is required to relate the field access which is known to be local within such call (say f) and the actual reference to it in the current context. This is done by using the access paths as follows. If a field f is local in S and it is always referenced through $l_i.f_1 \ldots f_n$, then when calling $q(\langle \bar{w} \rangle, \langle \bar{z} \rangle)$, the initial value of the corresponding ghost variable should be initialized to $w_i.f_1 \ldots f_n.f$. This is because l_i refers to the location to which the i-th argument points when calling q.

Example 15. Fig. 4 shows the transformed program for the declarations of Ex. 14. For simplicity, when a scope has only one version, we do not introduce new names for the corresponding procedures. Procedure *next* is not shown, it is as in Ex. 11. Procedure *hasNext* now incorporates a ghost variable v_s that tracks the value of the corresponding *state* field. Note that r calls $m{\cdot}\texttt{4}$ while q calls $m{\cdot}\texttt{5}$. For version 4 of m, both f and *state* are considered local and therefore we have the ghost variables v_s and v_f. Version 5 of m is not shown for lack of space, it is equivalent to version 4 but without any reference to v_f since it is not local in that context. Now, all methods can be proven terminating by using a field-insensitive analysis.

In practice, generating multiple versions for a given scope S might be expensive to analyze. However, two points should be noted: (1) when no accuracy is gained by the use of polyvariance, i.e., when the locality information is identical for all calling contexts, then the transformation behaves as monovariant; (2) when further accuracy is achieved by the use of polyvariance, a *context-sensitive*, but monovariant transformation can be preferred for efficiency reasons by simply declaring as local only those fields which are local in all versions.

$s(\langle x,y,z,w\rangle,\langle\rangle) \leftarrow$	$r_2(\langle x,y,z,w,s_0,v_f\rangle,\langle v_f\rangle)\leftarrow$	
$\underline{v_s{:=}y.state}$,	$\mathbf{s_0} \leq \mathbf{0}$.	$while{\cdot}4(\langle x,v_s,v_f\rangle,\langle v_s,v_f\rangle)\leftarrow$
$q(\langle y,w,z,v_s\rangle,\langle v_s\rangle)$,	$r_3(\langle x,y,z,w,s_0,v_f\rangle,\langle v_f\rangle)\leftarrow$	$hasNext(\langle x,v_s\rangle,\langle s_0,v_s\rangle)$,
$\underline{y.state{:=}v_s},\underline{v_f{:=}z.f}$,	$\mathbf{s_0} > \mathbf{10}, w{:=}x$,	$m_1{\cdot}4(\langle x,s_0,v_s,v_f\rangle,\langle v_s,v_f\rangle)$.
$r(\langle x,y,z,v_f\rangle,\langle v_f\rangle)$,	$r_4(\langle x,y,z,w,v_f\rangle,\langle v_f\rangle)$.	$m_1{\cdot}4(\langle x,s_0,v_s,v_f\rangle,\langle v_s,v_f\rangle)\leftarrow$
$\underline{z.f{:=}v_f}$.	$r_3(\langle x,y,z,w,s_0,v_f\rangle,\langle v_f\rangle)\leftarrow$	$\mathbf{s_0} \neq \mathsf{null}, next(\langle x,v_s\rangle,\langle s_0,v_s\rangle)$,
$q(\langle x,y,z,v_s\rangle,\langle v_s\rangle) \leftarrow$	$\mathbf{s_0} \leq \mathbf{10}, w{:=}y$,	$while{\cdot}4(\langle x,v_s,v_f\rangle,\langle v_s,v_f\rangle)$.
$m{\cdot}5(\langle x,y,z,v_s\rangle,\langle v_s\rangle)$.	$r_4(\langle x,y,z,w,v_f\rangle,\langle v_f\rangle)$.	$m_1{\cdot}4(\langle x,y,z,s_0,v_s,v_f\rangle,\langle v_s,v_f\rangle)\leftarrow$
$r(\langle x,y,z,v_f\rangle,\langle v_f\rangle)\leftarrow$	$r_4(\langle x,y,z,w,v_f\rangle,\langle v_f\rangle)\leftarrow$	$\mathbf{s_0} = \mathsf{null}$.
$w{:=}\mathsf{null}$,	$\underline{v_s{:=}x.state}$,	$hasNext(\langle this,v_s\rangle,\langle r,v_s\rangle)\leftarrow$
$r_1\langle x,y,z,w,v_f\rangle,\langle v_f\rangle)$.	$m{\cdot}4(\langle x,z,z,v_s,v_f\rangle,\langle v_s,v_f\rangle)$,	$s_0{:=}v_s$,
$r_1(\langle x,y,z,w,v_f\rangle,\langle v_f\rangle)\leftarrow$	$\underline{x.state{:=}v_s}$,	$hasNext_1(\langle this,s_0,v_s\rangle,\langle r,v_s\rangle)$.
$s_0{:=}z.f$,	$r_1(\langle x,y,z,w,v_f\rangle,\langle v_f\rangle)$.	$hasNext_1(\langle this,s_0,v_s\rangle,\langle r,v_s\rangle)\leftarrow$
$r_2(\langle x,y,z,w,s_0,v_f\rangle,\langle v_f\rangle)$.	$m{\cdot}4(\langle x,y,z,v_s,v_f\rangle,\langle v_s,v_f\rangle)\leftarrow$	$\mathbf{s_0} = \mathsf{null}, r{:=}0$.
$r_2(\langle x,y,z,w,s_0,v_f\rangle,\langle v_f\rangle)\leftarrow$	$while{\cdot}4(\langle x,v_s,v_f\rangle,\langle v_s,v_f\rangle)$,	$hasNext_1(\langle this,s_0,v_s\rangle,\langle r,v_s\rangle)\leftarrow$
$\mathbf{s_0} > \mathbf{0}, s_0{:=}z.f$,	$s_0{:=}v_f, s_0{:=}s_0-1, v_f{:=}s_0$,	$\mathbf{s_0} \neq \mathsf{null}, r{:=}1$.
$r_3(\langle x,y,z,w,s_0,v_f\rangle,\langle v_f\rangle)$.	$s_0{:=}v_f, s_0{:=}s_0-1, v_f{:=}s_0$.	

Fig. 4. Polyvariant Transformation of Running Example (excerpt)

7 Experiments

We have integrated our method in COSTA [4], a cost and termination analyzer for Java bytecode, as a pre-process to the existing field-insensitive analysis. It can be tried out at: `http://costa.ls.fi.upm.es`. The different approaches can be selected by setting the option enable_field_sensitive to: "trackable" for using the setting of [3]; "mono_local" for context-insensitive and monovariant transformation; and "poly_local" for context-sensitive and polyvariant transformation. In Table 1 we evaluate the precision and performance of the proposed techniques by analyzing three sets of programs. The first set contains loops from the JOlden suite [6] whose termination can be proven only by tracking reference fields. They are challenging because they contain reference-intensive kernels and use enumerators. The next set consists of the loops which access numeric field in their guards for all classes in the subpackages of "`java`" of SUN's J2SE 1.4.2. Almost all these loops had been proven terminating using the trackable profile [3]. Hence, our challenge is to keep the (nearly optimal) accuracy and comparable efficiency as [3]. The last set consists of programs which require a context-sensitive and polyvariant transformation (source code is available in the website above).

For each benchmark, we provide the size of the code to be analyzed, given as number of rules $\#\mathbf{R}$. Column $\#\mathbf{R}_p$ contains the number of rules after the polyvariant transformation which, as can be seen, increases only for the last set of benchmarks. Column $\#\mathbf{L}$ is the number of loops to be analyzed and $\#\mathbf{L}_p$ the same after the polyvariant transformation. Column $\mathbf{L}_{ins}$ shows the number of loops for which COSTA has been able to prove termination using a field-insensitive analysis. Note that, even if all entries correspond to loops which involve fields in their guards, they can contain inner loops which might not and, hence, can be proven terminating using a field-insensitive analysis. This is the case of many

Table 1. Accuracy and Efficiency of four Analysis Settings in COSTA

Bench.	**#R**	$\#\mathbf{R}_p$	**#L**	$\#\mathbf{L}_p$	$\mathbf{L}_{ins}$	$\mathbf{L}_{tr}$	$\mathbf{L}_{mono}$	$\mathbf{L}_{poly}$	$\mathbf{T}_{ins}$	$\mathbf{O}_{tr}$	$\mathbf{O}_{mono}$	$\mathbf{O}_{poly}$
bh	1759	1759	21	21	16	16	16	21	230466	1.54	1.25	1.50
em3d	1015	1015	13	13	1	3	3	13	17129	1.43	1.24	1.55
health	1364	1364	11	11	6	6	6	11	21449	2.23	1.65	2.00
java.util	593	593	26	26	3	24	24	24	17617	1.55	1.62	1.72
java.lang	231	231	14	14	5	14	13	13	2592	1.69	1.38	1.52
java.beans	113	113	3	3	0	3	3	3	3320	1.07	1.09	1.15
java.math	278	278	12	12	3	11	11	11	15761	1.07	1.05	1.12
java.awt	1974	1974	102	102	25	100	100	100	64576	1.25	1.21	1.55
java.io	187	187	4	4	2	4	4	4	2576	2.72	2.16	3.47
run-ex	40	61	2	3	0	0	1	3	300	1.28	1.25	2.24
num-poly	71	151	4	8	0	0	1	8	576	1.27	1.26	3.33
nest-poly	86	125	8	10	1	4	7	10	580	1.61	1.61	3.02
loop-poly	16	29	1	2	0	0	0	2	112	1.25	1.25	2.61

loops for benchmark bh. Columns $\mathbf{L}_{tr}$, $\mathbf{L}_{mono}$ and $\mathbf{L}_{poly}$ show the number of loops for which COSTA has been able to find a ranking function using, respectively, the trackable, mono_local and poly_local profiles as described above. Note that when using the poly_local option, the number of loops to compare to is $\#\mathbf{L}_p$ since the polyvariant transformation might increase the number of loops in $\#\mathbf{L}$.

As regards accuracy, it can be observed that for the benchmarks in the JOlden suite, trackable and mono_local behave similarly to a field-insensitive analysis. This is because most of the examples use iterators. Using poly_local, we prove termination of all of them. In this case, it can be observed from column $\#\mathbf{R}_p$ that context-sensitivity is required, however, the polyvariant transformation does not generate more than one version for any example. As regards the J2SE set, the profile trackable is already very accurate since these loops contain only numeric fields. Except for one loop in java.lang which involves several numeric fields, our profiles are as accurate as trackable. All examples in the last set include many reference fields and, as expected, the trackable does not perform well. Although mono_local improves the accuracy in many loops polyvariance is required to prove termination. The profile poly_local proves termination of all loops in this set.

We have tried to analyze the last set of benchmarks with the two other termination analyzers of Java bytecode publicly available, Julia [16] and AProVE [12]. Since polyvariance is required, Julia failed to prove termination of all of them. AProVE could not handle them because they use certain library methods not supported by the system. We have not been able to analyze the benchmarks in the Java libraries because the entries are calls to library methods which we could not specify as inputs to these systems. Something similar happens with the JOlden examples, the entries correspond to specific loops and we have not been able to specify them as input.

The next set of columns evaluate performance. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM. Column $\mathbf{T}_{ins}$

is the total time (in milliseconds) for field-insensitive analysis, and the other columns show the slowdown introduced by the corresponding field-sensitive analysis w.r.t. $\mathbf{T}_{ins}$. The overhead introduced by trackable and mono_local is comparable and, in most cases, is less than two. The overhead of poly_local is larger for the examples which require multiple versions and it increases with the size of the transformed program in $\#\mathbf{R}_p$. We argue that our results are promising since the overhead introduced is reasonable.

8 Conclusions and Related Work

Field sensitiveness is considered currently one of the main challenges in static analyses of object-oriented languages. We have presented a novel practical approach to field-sensitive analysis which handles all object fields (numeric and references) in a uniform way. The basic idea is to partition the program into fragments and convert object fields into local variables at each fragment, whenever such conversion is sound. The transformation can be guided by a context-sensitive analysis able to determine that an object field can be safely replaced by a local variable only for a specific context. This, when combined with a polyvariant transformation, achieves a very good balance between accuracy and efficiency.

Our work continues and improves over the stream of work on termination analysis of object-oriented bytecode programs [3,12,2,15,14]. The heuristic of treating fields as local variables in order to perform field-sensitive analysis by means of field-insensitive analysis was proposed by [3]. However, there are essential differences between both approaches. The most important one is that [3] handles only numeric fields and it is not effective to handle reference fields. A main problem is that [3] *replicates* numeric fields with equivalent local variables, instead of *replacing* them as we have to do to handle references as well, e.g., an instruction like $y.ref{:=}x$ is followed (i.e., replicated) by $v_{ref}{:=}x$. Replicating instructions, when applied to reference fields, makes $y.ref$ and v_{ref} alias, and therefore the path-length relations of v_{ref} will be affected by those of $y.ref$. In particular, further updates to $y.ref$ will force losing useful path-length information about v_{ref}, since the abstract field update (see [15]) loses valuable path-length information on anything that shares with y. Handling reference fields is essential in object-oriented programs, as witnessed in our examples and experimental results.

Techniques which rely on separation logic in order to track the depth (i.e., the path-length) of data-structures [5] would have the same limitation as path-length based techniques, if they are applied in a field-insensitive manner. This is because the depth of a variable x does not necessarily decrease when the depth of one of its field decreases (see Sec. 2). However, by applying these techniques on our transformed programs, we expect them to infer the required information without any modification to their analyses.

Acknowledgements. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the

Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the TIN2008-04473-E (Acción Especial) project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. Aiken, A., Foster, J.S., Kodumal, J., Terauchi, T.: Checking and inferring local non-aliasing. In: Proc. of PDLI 2003, pp. 129–140. ACM, New York (2003)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Field-Sensitive Value Analysis by Field-Insensitive Analysis. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 370–386. Springer, Heidelberg (2009)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Resource usage analysis and its application to resource certification. In: FOSAD 2007. LNCS, vol. 5705, pp. 258–288. Springer, Heidelberg (2009)
5. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
6. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in Java. In: IEEE PaCT, pp. 280–291 (2001)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. POPL. ACM, New York (1978)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
9. Lehner, H., Müller, P.: Formal translation of bytecode into BoogiePL. In: Bytecode 2007. ENTCS, pp. 35–50. Elsevier, Amsterdam (2007)
10. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1996)
11. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: LCTES (2006)
12. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Termination Analysis of Java Bytecode by Term Rewriting. In: Waldmann, J. (ed.) International Workshop on Termination, WST 2009, Leipzig, Germany (June 2009)
13. Puebla, G., Hermenegildo, M.: Abstract Multiple Specialization and its Application to Program Parallelization. JLP 41(2&3), 279–316 (1999)
14. Rossignoli, S., Spoto, F.: Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 95–110. Springer, Heidelberg (2005)
15. Spoto, F., Hill, P., Payet, E.: Path-length analysis of object-oriented programs. In: EAAI 2006. ENTCS. Elsevier, Amsterdam (2006)
16. Spoto, F., Mesnard, F., Payet, É.: A Termination Analyser for Java Bytecode based on Path-Length. ACM TOPLAS (2010) (to appear)
17. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999)
18. Xia, S., Fähndrich, M., Logozzo, F.: Inferring Dataflow Properties of User Defined Table Processors. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 19–35. Springer, Heidelberg (2009)

Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs

Christophe Alias[1], Alain Darte[1], Paul Feautrier[1], and Laure Gonnord[2]

[1] Compsys team, LIP, Lyon, France
UMR 5668 CNRS—ENS Lyon—UCB Lyon—Inria
{Firstname.Lastname}@ens-lyon.fr
[2] LIFL - UMR CNRS/USTL 8022, INRIA Lille - Nord Europe
40 avenue Halley, 59650 Villeneuve d'Ascq, France
Laure.Gonnord@lifl.fr

Abstract. Proving the termination of a flowchart program can be done by exhibiting a ranking function, i.e., a function from the program states to a well-founded set, which strictly decreases at each program step. A standard method to automatically generate such a function is to compute invariants for each program point and to search for a ranking in a restricted class of functions that can be handled with linear programming techniques. Previous algorithms based on affine rankings either are applicable only to simple loops (i.e., single-node flowcharts) and rely on enumeration, or are not complete in the sense that they are not guaranteed to find a ranking in the class of functions they consider, if one exists. Our first contribution is to propose an efficient algorithm to compute ranking functions: It can handle flowcharts of arbitrary structure, the class of candidate rankings it explores is larger, and our method, although greedy, is provably complete. Our second contribution is to show how to use the ranking functions we generate to get upper bounds for the computational complexity (number of transitions) of the source program. This estimate is a polynomial, which means that we can handle programs with more than linear complexity. We applied the method on a collection of test cases from the literature. We also show the links and differences with previous techniques based on the insertion of counters.

1 Introduction and Motivation

The problem of proving program correctness has been with us since the early days of Computer Science. In a seminal paper [20], R. W. Floyd proposed what has become one of the standard approaches: affix assertions to each program point and prove that they are consequences of the assertions of its predecessors in the program control graph. The assertions at the entry point of the program are its *preconditions*, the assertions at loop entry points are *invariants*, while the assertions at its exit point must entail correctness, according to some set of requirements. Constructing the required set of assertions is a tedious and error-prone task. The automatic construction of invariants has been proved to be intractable in the general case [6]. However, partial or conservative solutions can be obtained by abstract interpretation methods [14].

At the same time, it was soon realized that this method proves only partial correctness, i.e., that the program gives the correct result if and when it terminates. To prove

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 117–133, 2010.
© Springer-Verlag Berlin Heidelberg 2010

termination, one needs a variant or ranking function (a W-function in Floyd's terminology), i.e., a function from the states of the program to some well-founded set, which strictly decreases at each program step. Of course, designing an algorithm for building ranking functions in all cases is not possible since it would give a solution to the undecidable halting problem. However, this does not preclude the existence of partial solutions, which, e.g., handle only programs (or approximated models) of a restricted shape, or look for rankings in a restricted class of functions. Our first contribution is to generalize previous work for generating ranking functions. We design an algorithm with the following features:

- It can handle flowcharts of arbitrary structure.
- The class of rankings we consider is much larger: in the global ranking function we generate, each program point can have its own multi-dimensional affine expression.
- Our algorithm is based on a greedy mechanism. Nevertheless, our technique is provably complete, even for our larger class of ranking functions.

There are many variations on the above theme. For instance, as in [27], one may select a set of *cutpoints*, with the property that their removal makes the flowchart acyclic. It is then enough to exhibit a function, non increasing everywhere, that decreases and is well-founded at each cutpoint. One may even proceed each flowchart cycle at a time.

Our second contribution is to show that the global ranking functions we generate can be used to give upper bounds on the worst-case computational complexity (WCCC) of the program execution, i.e., the number of transitions that can be made in an execution trace. Obviously, if a program does not terminate, its WCCC is infinite. If the program terminates and a one-dimensional ranking function exists, its value at program start is an upper bound on the number of steps before termination since it decreases at least by one at each program step. The situation is more complicated in the case of multi-dimensional ranking functions but we show how the WCCC can be computed thanks to counting techniques in polyhedra. Furthermore, our ranking algorithm has an additional important feature: It generates a multi-dimensional affine ranking function whose dimension is minimal. This is important to get an accurate upper bound on the WCCC of the flowchart program. To the best of our knowledge, our technique is the first one that uses ranking functions to compute upper bounds on the number of iterations of arbitrary loops (a particular case of the WCCC).

The rest of the paper is organized as follows. Section 2 gives some basic notations and concepts: the program abstraction we use (*integer interpreted automata*) and the class of ranking functions we consider. Section 3 presents our method for constructing multi-dimensional affine ranking functions and states its completeness. Section 4 explains how we infer the computational complexity of the source program. Section 5 reports on our implementation through a collection of benchmarks from the literature. Section 6 describes other approaches to the termination problem and WCCC evaluation. We then conclude pointing to some unsolved problems and outlining future work.

2 Notations and Definitions

We write matrices with capital letters (as A) and column vectors in boldface (as $\boldsymbol{x}$). If $\boldsymbol{x}$ has dimension d, its components are denoted $\boldsymbol{x}[i]$, with $0 \leq i < d$. Thus, its i-th component is $\boldsymbol{x}[i-1]$. Sets are represented with calligraphic letters such as $\mathcal{W}$, $\mathcal{K}$, etc.

2.1 Integer Interpreted Automata

In the tradition of most previous work on program termination and static analysis, we first transform the program to be analyzed into an abstraction: the associated *integer interpreted automaton*. This is similar to the flowcharts used a long time ago to express programs (see, e.g., Manna's book [27]) until the advent of structured programming. In fact, when one looks at real-life programs, many deviations from the strict structured model can occur, including premature loop termination, exceptions, and even the occasional `goto`. Reasoning with flowcharts abstracts the details of the syntax and semantics of the source language, which can be dealt with by an appropriate preprocessor.

In our work, a program is represented by an *affine (integer) interpreted automaton* $(\mathcal{K}, n, k_{init}, \mathcal{T})$ defined by:

- a finite set $\mathcal{K}$ of *control points*;
- n integer variables represented by a vector $\boldsymbol{x}$ of size n;
- an initial control point $k_{init} \in \mathcal{K}$;
- a finite set $\mathcal{T}$ of 4-tuples (k, g, a, k'), called *transitions*, where $k \in \mathcal{K}$ (resp. $k' \in \mathcal{K}$) is the source (resp. target) control point, $g : \mathbb{Z}^n \mapsto \mathbb{B} = \{\text{true}, \text{false}\}$, the *guard*, is a logical formula expressed with affine inequalities $G\boldsymbol{x} + \boldsymbol{g} \geq 0$, and $a : \mathbb{Z}^n \mapsto \mathbb{Z}^n$, the *action*, assigns, to each variable valuation $\boldsymbol{x}$, a vector $\boldsymbol{x}'$ of size n, expressed by an affine expression $\boldsymbol{x}' = A\boldsymbol{x} + \boldsymbol{a}$. Here, G and A are matrices, $\boldsymbol{g}$ and $\boldsymbol{a}$ are vectors.

To represent non-determinism or to approximate non-affine or non-analyzable assignments in the program, we may have to assign the value "?", representing an arbitrary integer, to a variable, but we will not elaborate on this point. This is equivalent to deal with affine relations between $\boldsymbol{x}$ and $\boldsymbol{x}'$ instead of functions, see [1] for details.

Semantics. The set of states is $\mathcal{K} \times \mathbb{Z}^n$. A *trace* from $(k_0, \boldsymbol{x}_0)$ to $(k, \boldsymbol{x})$ is a sequence $(k_0, \boldsymbol{x_0}), (k_1, \boldsymbol{x}_1), \ldots, (k_p, \boldsymbol{x}_p)$ such that $k_p = k$, $\boldsymbol{x}_p = \boldsymbol{x}$ and for each i, $0 \leq i < p$, there exists in $\mathcal{T}$ a transition (k_i, g_i, a_i, k_{i+1}) such that $g_i(\boldsymbol{x}_i) = \text{true}$ and $\boldsymbol{x}_{i+1} = a_i(\boldsymbol{x}_i)$. Given an initial valuation $\boldsymbol{v}$, a state $(k, \boldsymbol{x})$ is *reachable from* $\boldsymbol{v}$ iff (if and only if) there is a trace from $(k_{init}, \boldsymbol{v})$ to $(k, \boldsymbol{x})$. A state $(k, \boldsymbol{x})$ is *reachable* if there exists $\boldsymbol{v} \in \mathbb{Z}^n$ such that $(k, \boldsymbol{x})$ is reachable from $\boldsymbol{v}$. The set of reachable states is denoted by $\mathcal{R}$.

Invariants. The guard g in a transition $t = (k, g, a, k')$ gives a necessary condition on variables $\boldsymbol{x}$ to traverse the transition t and to apply its corresponding action a. To get the exact valuations $\boldsymbol{x}$ of variables for which the action a can be performed, one would need to take into account the initial valuations and the successive conditions that led to the control point k. We denote by $\mathcal{R}_k$ the set of possible valuations $\boldsymbol{x}$ of variables when the control is in k:

$$\mathcal{R}_k = \{\boldsymbol{x} \in \mathbb{Z}^n \mid (k, \boldsymbol{x}) \in \mathcal{R}\}.$$

Then, there exists a trace containing the transition (k, g, a, k') iff $\boldsymbol{x} \in \mathcal{R}_k$ and $g(\boldsymbol{x})$ is true. Note that $\mathcal{R}_k$ does not depend on any initial valuation. More precisely, it is the union, for all initial valuations $\boldsymbol{v}$, of the set of vectors $\boldsymbol{x}$ such that $(k, \boldsymbol{x})$ is reachable from $\boldsymbol{v}$.

In practice, it is difficult to determine the set $\mathcal{R}_k$ exactly but it is possible to give over-approximations, thanks to the notion of *invariants*. An invariant on a control point k is a formula $\phi_k(\boldsymbol{x})$ that is true for all reachable states $(k, \boldsymbol{x})$. It is *affine* if it is the conjunction

of a finite number of affine conditions on program variables. The set $\mathcal{R}_k$ is then over-approximated by the integer points within a polyhedron $\mathcal{P}_k$. To compute invariants, we rely on standard *abstract interpretation* techniques, widely studied since the seminal paper of Cousot and Halbwachs [14]. These sets $\mathcal{P}_k$ represent all the information on the values of variables that can be deduced from the program by state-of-the-art analysis techniques. Unlike [8,24] where the construction of invariants is coupled with the termination proof or evaluation of iteration bounds, the invariants $\mathcal{P}_k$ are pre-computed and are the inputs of the techniques developed in the next sections.

2.2 Termination and Ranking Functions

Invariants can only prove partial correctness of a program. The standard technique for proving termination is to consider ranking functions to well-founded sets. A well-founded set $\mathcal{W}$ is a set with a (total or partial) order $\preceq$ (we write $a \prec b$ if $a \preceq b$ and $a \neq b$) such that there is no infinite descending chain, i.e., no infinite sequence $(x_i)_{i\in\mathbb{N}}$ with $x_i \in \mathcal{W}$ and $x_{i+1} \prec x_i$ for all $i \in \mathbb{N}$.

Definition 1. *A* ranking *is a function $\rho : \mathcal{K} \times \mathbb{Z}^n \to \mathcal{W}$, from the automaton states to a well-founded set $(\mathcal{W}, \preceq)$, whose values decrease at each transition $t = (k, g, a, k')$:*

$$\boldsymbol{x} \in \mathcal{R}_k \wedge g(\boldsymbol{x}) = true \wedge \boldsymbol{x}' = a(\boldsymbol{x}) \Rightarrow \rho(k', \boldsymbol{x}') \prec \rho(k, \boldsymbol{x}) \tag{1}$$

It is said affine *if it is affine in the second parameter (the variables).*

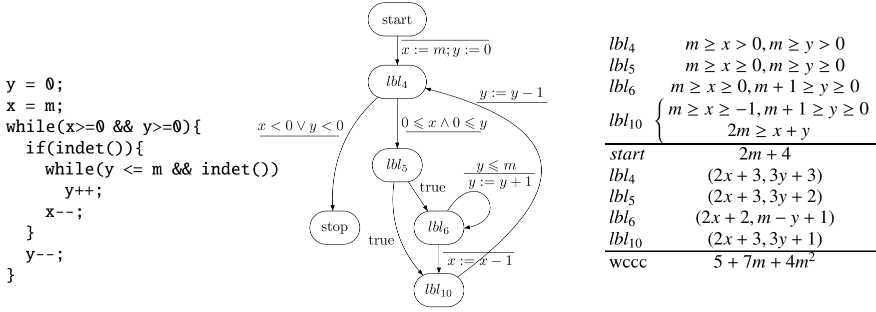
Definition 2. *A ranking function is* one-dimensional *if its co-domain is $(\mathbb{N}, \leq)$. It is k-*dimensional *(or multi-dimensional of dimension k) if its co-domain is $(\mathbb{N}^k, \preceq_k)$, where the order $\preceq_k$ is the standard lexicographic order on integer vectors.*

Obviously, the existence of a ranking function implies program termination for any valuation $\boldsymbol{v}$ at the initial control point k_{init}. A well-known property is that an integer interpreted automaton terminates for any initial valuation if and only if it has a ranking function. Furthermore, if it terminates and has bounded non-determinism, there is a one-dimensional ranking function, which is not necessarily affine.

2.3 Illustrating Example

An example program is given in Fig. 1, with its corresponding automaton. The control points are labelled for convenience, and transitions are depicted with arrows indexed by $\frac{g}{a}$ (g is omitted when g = true). State names are assigned arbitrarily by our parser.

The C code features two nested loops, which do not fit into the structured programming model, since the inner counter, `y`, is modified in the outer loop. The `indet` function abstracts non-determinism or an intractable test. The outcome of non-determinism is that, in the corresponding automaton, both transitions out of state lbl_5 have a true guard. The right of Fig. 1 successively gives, assuming $m > 0$, the invariants as found by Aspic (an abstract-interpretation based invariant generator, see Section 5), followed by the bidimensional rankings and the corresponding WCCC computed by Rank, our tool. The reader may care to check that these rankings are positive and lexicographically decrease along each transition. For instance, the first component of the ranking function decreases from $2x + 3$ at lbl_5 to $2x + 2$ at lbl_6, then $2x + 3$ at lbl_{10}, but since x is changed to $x - 1$ by the corresponding transition, the ranking has really decreased.




```
y = 0;
x = m;
while(x>=0 && y>=0){
  if(indet()){
    while(y <= m && indet())
      y++;
    x--;
  }
  y--;
}
```

lbl_4	$m \geq x > 0, m \geq y > 0$
lbl_5	$m \geq x \geq 0, m \geq y \geq 0$
lbl_6	$m \geq x \geq 0, m+1 \geq y \geq 0$
lbl_{10}	$\begin{cases} m \geq x \geq -1, m+1 \geq y \geq 0 \\ 2m \geq x+y \end{cases}$
start	$2m+4$
lbl_4	$(2x+3, 3y+3)$
lbl_5	$(2x+3, 3y+2)$
lbl_6	$(2x+2, m-y+1)$
lbl_{10}	$(2x+3, 3y+1)$
wccc	$5+7m+4m^2$

Fig. 1. Illustrating example

3 Computing Affine Ranking Functions

This section gives an algorithm to build a multi-dimensional affine ranking function, i.e., a ranking function $\rho : \mathcal{K} \times \mathbb{Z}^n \to \mathbb{N}^d$, affine for the second parameter. The integer d is the dimension of the ranking. Considering ranking functions with $d > 1$ is mandatory to be able to prove the termination of programs that induce a number of transitions, i.e., a trace length, more than linear in the program parameters. Furthermore, when a d-dimensional ranking exists, the number of transitions can be bounded by a polynomial, derived from the ranking, with a simpler method than by manipulating directly polynomials of degree d. Considering rankings with a different affine function for each control point also extends the set of programs whose termination can be determined, compared for example to the technique of [13] (see more details in Section 3.2).

3.1 A Greedy Polynomial-Time Procedure

As explained in Section 2.1, in practice, the exact sets $\mathcal{R}_k$ are not necessarily available. They are over-approximated by invariants $\mathcal{P}_k$, with $\mathcal{R}_k \subseteq \mathcal{P}_k$, which are, in our case, described by polyhedra. The conditions that a ranking function must satisfy are then related to these invariants and not to the exact sets of reachable states.

A ranking function ρ of dimension d needs to satisfy two properties. First, as ρ has co-domain $\mathbb{N}^d$, it should assign a nonnegative integer vector to each relevant state:

$$\boldsymbol{x} \in \mathcal{P}_k \Rightarrow \rho(k, \boldsymbol{x}) \geq \mathbf{0} \text{ (component-wise)} \tag{2}$$

Second, it should decrease on transitions. Let Q_t be the polyhedron described by the constraints of a transition $t = (k, g, a, k')$, i.e., $\boldsymbol{x} \in \mathcal{P}_k$, $g(\boldsymbol{x})$ is true, and $\boldsymbol{x}' = a(\boldsymbol{x})$, which can be built from matrices A and G, and vectors $\boldsymbol{a}$ and $\boldsymbol{g}$ (see Section 2.1). For an automaton whose actions are general affine *relations*, Q_t is directly given by the action definitions. With $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}') = \rho(k, \boldsymbol{x}) - \rho(k', \boldsymbol{x}')$, Inequality (1) then becomes:

$$(\boldsymbol{x}, \boldsymbol{x}') \in Q_t \Rightarrow \Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}') \succ_d \mathbf{0} \tag{3}$$

which means $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}') \neq \mathbf{0}$ and its first nonzero component is positive. It this component is the i-th, the *level* of $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}')$ is i. A transition t is said to be (fully) *satisfied by the i-th component* of ρ (or *at dimension* i) if the maximal level of all $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}')$ is i.

To build a ranking ρ, the difficulty is to decide, for each transition t and for each pair $(\boldsymbol{x}, \boldsymbol{x}') \in Q_t$, what will be the level of $\Delta_t(\rho, \boldsymbol{x}, \boldsymbol{x}')$ and by which component of ρ the transition t will be satisfied. A potentially exponential search, as in [8], should be avoided. To address this issue, our algorithm uses the same greedy mechanism as in [25,19,13]. The components of ρ are functions from $\mathcal{K} \times \mathbb{Z}^n$ to $\mathbb{N}$. We build them, one after the other, from the first one to the last one. For a component σ of ρ and a transition t not yet satisfied by one of the previous components of ρ, we consider the constraint:

$$(\boldsymbol{x}, \boldsymbol{x}') \in Q_t \Rightarrow \Delta_t(\sigma, \boldsymbol{x}, \boldsymbol{x}') \geq \varepsilon_t \text{ with } 0 \leq \varepsilon_t \leq 1. \tag{4}$$

and we select a ranking such that as many transitions as possible have $\varepsilon_t = 1$, i.e., are now satisfied. Surprisingly, despite this *greedy* approach, our technique is provably complete (see Theorem 1), which means that if a multi-dimensional affine ranking exists, our algorithm finds one. Our algorithm can then be summarized as follows:

```
1: i = 0; T = 𝒯;                                          ▹ Initialize T to the set of all transitions
2: while T is not empty do
3:     Find a 1D affine function σ and values ε_t such that all inequalities (2) and (4) are satisfied
   and as many ε_t as possible are equal to 1;             ▹ This means maximizing ∑_{t∈T} ε_t
4:     Let ρ_i = σ ; i = i + 1;                             ▹ σ defines the i-th component of ρ
5:     If no transition t with ε_t = 1, return false        ▹ No multi-dimensional affine ranking.
6:     Remove from T all transitions t such that ε_t = 1;   ▹ The transitions have level i
7: end while;
8: d = i; return true;                                      ▹ There is a d-dimensional ranking
```

For Line 3, any solution σ leading to $\varepsilon_t > 0$ can be multiplied by a suitable positive constant to get a solution with $\varepsilon_t = 1$. Thus, for any solution maximizing $\sum_{t \in T} \varepsilon_t$, a transition t has either $\varepsilon_t = 0$ or $\varepsilon_t = 1$. At each iteration of the while loop, σ is used as a new component of the ranking ρ (Line 4). By construction, ρ is strictly decreasing at this level for all transitions t with $\varepsilon_t = 1$. No need to consider them any longer, which means that they are removed for building subsequent components (Line 6). If no transition is removed, no ranking function is derived and the automaton may not terminate.

To find a suitable function σ at Line 3, we use linear programming. The set of inequalities that we need to solve are Inequalities (2) (with σ instead of ρ) and (4). The standard method (used in [19,28,8]) is to rely on the affine form of Farkas lemma [30]:

Lemma 1 (Farkas lemma, affine form). *An affine form $\phi : \mathbb{R}^n \to \mathbb{R}$ with $\phi(\boldsymbol{x}) = \boldsymbol{c}.\boldsymbol{x} + c_0$ is nonnegative everywhere in a non-empty polyhedron $\{\boldsymbol{x} \mid A\boldsymbol{x} + \boldsymbol{a} \geq \boldsymbol{0}\}$ iff:*

$$\exists \boldsymbol{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \phi(\boldsymbol{x}) \equiv \boldsymbol{\lambda}.(A\boldsymbol{x} + \boldsymbol{a}) + \lambda_0$$

The notation $\equiv$ is a formal equality, which means that $\boldsymbol{x}$ can be eliminated and coefficients identified. In other words:

$$\exists \boldsymbol{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \boldsymbol{c} = \boldsymbol{\lambda}.A \text{ and } c_0 = \boldsymbol{\lambda}.\boldsymbol{a} + \lambda_0$$

We can now apply the affine form of Farkas lemma to Inequalities (2) (with σ instead of ρ) and (4). With $\mathcal{P}_k = \{\boldsymbol{x} \mid P_k\boldsymbol{x} + \boldsymbol{p_k} \geq \boldsymbol{0}\}$, we transform Inequality (2) into:

$$\exists \boldsymbol{\lambda}_k \in (\mathbb{R}^+)^n, \lambda_k^0 \in \mathbb{R}^+ \text{ such that } \sigma(k, \boldsymbol{x}) \equiv \boldsymbol{\lambda}_k.(P_k\boldsymbol{x} + \boldsymbol{p_k}) + \lambda_k^0 \tag{5}$$

Similarly, with $\mathcal{Q}_t = \{\boldsymbol{y} = (\boldsymbol{x}, \boldsymbol{x}') \mid Q_t\boldsymbol{y} + \boldsymbol{q}_t \geq \boldsymbol{0}\}$, we transform Inequality (4) into:

$$\exists \boldsymbol{\mu}_t \in (\mathbb{R}^+)^n, \mu_t^0 \in \mathbb{R}^+ \text{ s.t. } \Delta_t(\sigma, \boldsymbol{x}, \boldsymbol{x}') - 1 \equiv \boldsymbol{\mu}_t.(Q_t\boldsymbol{y} + \boldsymbol{q}_t) + \mu_t^0 \quad (6)$$

A substitution of (5) in (6) and an identification on each dimension of $\boldsymbol{y}$ leads to a set of linear inequalities. Considering all inequalities obtained for all transitions $t \in T$ and maximizing $\sum_{t \in T} \varepsilon_t$ (Line 3 of the algorithm) leads to the desired function σ.

Note: As we use linear programming, but not integer linear programming, we may end up with a function σ with rational components. However, we can always multiply it by a suitable integer to get a ranking function with integer values.

Example of Section 2.3 (Cont'd). Write $\sigma_k(x, y) = a_k x + b_k y + c_k$ the 1st component of the ranking. Consider any transition, e.g., $lbl_4 \to lbl_5$. The non-increasing constraint gives $(a_4 - a_5)x + (b_4 - b_5)y + c_4 - c_5 \geq 0$. Letting $x = 0$ and $y = m$, and noticing that m can be arbitrarily large, gives $b_4 \geq b_5$. The same technique applied to all transitions of a cycle shows that all b_k (same for all a_k) of a strongly connected component are equal: let b this value. For the self-loop on lbl_6, $\sigma_6(x, y) \geq \sigma_6(x, y+1)$ implies $b \leq 0$. The cycle $lbl_4 \to lbl_5 \to lbl_{10} \to lbl_4$ implies $\sigma_4(x, y) \geq \sigma_4(x, y-1)$, thus $b \geq 0$. Hence, these two cycles cannot be satisfied at the first dimension. However, the transitions $lbl_5 \to lbl_6$ and $lbl_6 \to lbl_{10}$ can be satisfied, disconnecting the two cycles and allowing them to be satisfied separately by the 2nd component of ρ. Here, we have deliberately simplified the problem by ignoring the positivity constraints and using qualitative reasoning for analyzing the descent constraints. In our tool, linear programming replaces intuition.

3.2 Completeness

Since non-terminating programs exist, there is no hope of proving that a ranking function always exists. Moreover, there are terminating affine interpreted automata with no multi-dimensional affine ranking. Thus, all we can prove is that, if a multi-dimensional affine ranking exists, our algorithm finds one, i.e., it is *complete* for the class of multi-dimensional affine rankings. Also, as the sets $\mathcal{R}_k$ are over-approximated by the invariants $\mathcal{P}_k$, completeness has to be understood with respect to these invariants, which means that if the algorithm fails when an affine ranking exists, it is because invariants are not accurate enough. In this section, we just sketch the completeness proof. The proof itself, quite long and technical, can be found in the long version of this paper [2].

Theorem 1. *If an affine interpreted automaton, with associated invariants, has a multi-dimensional affine ranking function, then the algorithm of Section 3.1 finds one. Moreover, the dimension of the generated ranking is minimal.*

There can be several reasons why a greedy algorithm could be incomplete. First, we could make a bad choice when selecting the transitions that are satisfied at a given dimension. However, there is no decision to make: if two transitions can be satisfied, one by a function σ_1, the other by a function σ_2, both can be simultaneously satisfied by the function $\sigma_1 + \sigma_2$. Second, enforcing that each transition is satisfied at the smallest possible dimension could also be a bad decision. Third, keeping all pairs $(\boldsymbol{x}, \boldsymbol{x}')$ in Inequality (4) until the transition is fully satisfied, even those for which the ranking is

decreasing for a previous dimension, could overconstrain the problem too. In particular, asking that at least one transition is (fully) satisfied at each dimension (Line 5 of the algorithm) could be too demanding. One could imagine situations where all transitions are partially satisfied, but none is fully satisfied. Theorem 1 shows that this is not the case. Despite all these greedy choices, the completeness is not lost.

To summarize the proof, we start from an affine ranking of dimension d, if one exists. We show that there is an affine ranking of dimension d that fully satisfies at least one transition. This proves that our algorithm does not abort and generates a one-dimensional ranking σ. Then, we show that there is an affine ranking of dimension d whose first component is σ. Finally, we show that there is an affine ranking of dimension d, whose first component is σ, and such that the $d-1$ last components satisfy all transitions not fully satisfied by σ. Iterating the process, this shows our algorithm terminates and generates an affine ranking of dimension $\leq d$, for any possible dimension d.

The knowledgeable reader may have noticed a similarity with the algorithm of [13]. However, as pointed out earlier, the class of ranking functions we consider is larger. In our case, at each step of the construction, one component (i.e., dimension) σ of the global ranking function ρ is defined, and each control point k can have a different affine expression: $\sigma(k, \boldsymbol{x}) = \boldsymbol{\lambda}_k.\boldsymbol{x}+c_k$, where $\boldsymbol{\lambda}_k$ is a vector and c_k a scalar. The algorithm in [13] proceeds differently. At each step of the construction, instead of building a global ranking function, it checks, for each transition, if there exists an affine expression decreasing for this transition and non-increasing for all other transitions of the same strongly connected component (SCC). All transitions for which this is possible are removed as well as transitions that now do not belong to any SCC. One can prove that if this technique succeeds, there is also a component σ, which is decreasing for all removed transitions, non-increasing for other, of the form $\sigma(k, \boldsymbol{x}) = \boldsymbol{\lambda}.\boldsymbol{x} + c_k$, in other words a unique linear part for all control points, plus some shifts (the c_k), exactly as the loop scheduling technique of [18]. Such a restricted form is particularly useful when the automaton actions define simple translations, as for the example of Section 2.3, because Farkas lemma is then not needed. However, as the following examples show, this class of functions is less powerful than general affine rankings. In other words, the algorithm of [13] is not complete with respect to the class of all multi-dimensional affine rankings.

In the synthetic examples of Figure 2, to make the discussion simpler, we selected the lower bounds for x and y so that these two variables are always nonnegative. The two examples have then similar ranking functions: $2 + 3m$ and 0 for the start and stop program points, and $2x + y + 1$ for k_1, $x + y + 1$ for k_2 and k_3 (for the second example). They are thus proved to terminate with $O(m)$ transitions in any execution trace. If the same linear part is chosen in each SCC as previously explained (or equivalently if the technique of [13] is applied to prove termination), the result is not as accurate. The first component of the ranking cannot depend on y (due to the potentially-parametric increases and decreases of y on the two transitions between k_1 and k_2), it is thus a function of x only. For the first example, a two-dimensional ranking is generated (we get $x+1$ for k_1 and $(x+1, y)$ for k_2), thus the program is still proved to terminate but appears to have a quadratic complexity. For the second example however, as x decreases on the two transitions between k_1 and k_2, but increases on the self-loop on k_3, no transition can be satisfied at the first dimension, and the technique fails to prove termination.

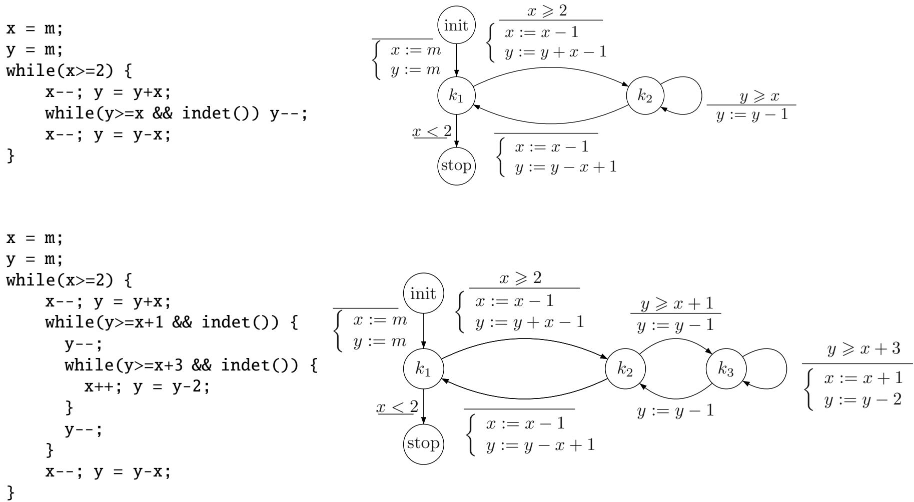


Fig. 2. Examples requiring general affine ranking functions

4 Worst-Case Computational Complexity (WCCC)

As shown in the survey by Wilhelm et al. [33], the computation of a worst-case execution time (WCET) is a highly complex affair, as it has to take into account the program, its data, and the processor on which it runs. Handling all these complexities is beyond the scope of this paper. Our aim is to evaluate an *abstract* WCET, as would be observed on a processor with a perfectly additive timing model, executing one automaton transition in unit time. We call this quantity the *worst-case computational complexity* of the program (WCCC). Such an estimate can be useful, for example as a template with unknown coefficients, to be fitted to actual measurements by a process of regression. It is also standard in high-level synthesis to need an upper-bound on the number of loop iterations (`do` loops as well as `while` loops), to enable scheduling optimizations at higher levels. We thus define the WCCC as an upper bound on the number of transitions executed, given an initial value of the counter variables. Note that the WCCC is significant only up to constant factors. For example, if we eliminate a state by edge coalescing, the semantics of the flowchart will not be materially changed, but the WCCC may decrease.

With this definition, one could over-approximate the WCCC of a terminating program by the total number of reachable states (because a finite trace cannot contain twice the same state), i.e., WCCC $\leq \sum_k \#\tilde{\mathcal{R}}_k$ or even more conservatively WCCC $\leq \sum_k \#\tilde{\mathcal{P}}_k$ as $\mathcal{R}_k$ is itself over-approximated by $\mathcal{P}_k$. [1] This is a very rough over-approximation but, even worse, this technique can lead to an infinite WCCC, even for a terminating automaton, if some invariant $\mathcal{P}_k$ is unbounded. Rather, we can use the ranking function itself to prune the invariant sets. Indeed, consider a trace $(k_0, \boldsymbol{x}_0)$, ..., $(k_p, \boldsymbol{x}_p)$ in the execution of the automaton. By definition of a ranking function, $\rho(k_{i+1}, \boldsymbol{x}_{i+1}) \prec \rho(k_i, \boldsymbol{x}_i)$. Since $\prec$ is a strict order, it follows by transitivity that all $\rho(k_i, \boldsymbol{x}_i)$ are distinct in $\mathcal{W}$.

[1] Here, the notation $\tilde{\mathcal{S}}$ means the integral points in a set $\mathcal{S}$, and $\#\tilde{\mathcal{S}}$ denotes the cardinal of $\tilde{\mathcal{S}}$.

Hence, the length of the trace is bounded by the cardinal of the co-domain of ρ:

$$\text{WCCC} \le \# \bigcup_k \rho(k, \tilde{\mathcal{P}}_k) \le \sum_k \# \rho(k, \tilde{\mathcal{P}}_k) \tag{7}$$

The first inequality is more accurate but harder to compute as it involves a union of sets. So far, in our implementation, we use the second less accurate inequality.

Let us see how we can compute $\#\rho(k, \tilde{\mathcal{P}}_k)$ for a given control point k. To make notations simpler, we drop the index k: we let $\rho(k, \boldsymbol{x}) = \rho(\boldsymbol{x}) = R\boldsymbol{x} + \boldsymbol{r}$ and $\mathcal{P} = \mathcal{P}_k$. To compute $\#\rho(\tilde{\mathcal{P}})$, we can ignore the constant vector $\boldsymbol{r}$. The number of different values in $\rho(\tilde{\mathcal{P}})$ is then the number of points in the image of a $\mathbb{Z}$-polyhedron (intersection of an integral lattice, here $\mathbb{Z}^n$, and a polyhedron, here $\mathcal{P}$) by an affine function. If R is injective, it is of course equal to the number of integral points in the invariant itself. Otherwise, there are three issues: the fact that several points can have the same image (thus the kernel of the mapping must be identified), the fact that some regular holes can arise (sub-lattice of $\mathbb{Z}^n$) in the image of the polyhedron, and the fact that some irregular holes can appear on its boundaries. Such problems have been widely studied in the literature using various techniques related to Ehrhart polynomials [11,31]. So far, we implemented a simpler over-approximation method, which normalizes R in such a way that $\rho(\tilde{\mathcal{P}})$ no longer contains regular holes. This way, a standard computation of integral points can be applied. This normalization is done thanks to the Smith normal form. We compute $R = USV$, where U and V are unimodular, $S = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$, and D is a diagonal positive matrix of rank d (the rank of R). We compute $\mathcal{V}$ the polyhedron obtained by projecting the polyhedron $V\mathcal{P} = \{V\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{P}\}$ on its d first components. Actually, $\#\tilde{\mathcal{V}}$ is a slight over-approximation of $\#\rho(\tilde{\mathcal{P}})$. Indeed, two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\tilde{\mathcal{P}}$ have the same image by ρ if and only if $V\boldsymbol{x}$ and $V\boldsymbol{y}$ have the same d first components. The over-approximation comes from the fact that, in very specific cases, not all integral vectors in $\mathcal{V}$ are obtained by projection of an integral vector in $V\mathcal{P}$. The number of integral vectors in $\mathcal{V}$ is then computed using Ehrhart polynomials.

It is important to minimize the rank of D because the WCCC will tend to be smaller if the dimension of $\mathcal{V}$ is smaller. This is why it is important to generate rankings of minimal dimension as our algorithm does (Theorem 1). However, adding linearly-dependent components to the ranking will simply add null rows at the bottom of the matrix S. From this follows that the WCCC will be $O(M^n)$, if M is an upper bound for all variables, since it is impossible to build more than n linear forms on n variables. This bound cannot be improved, since with n variables, one can write a system of n perfectly nested loops, which achieves the required complexity.

The factors affecting the precision of the WCCC, beside the union computation, are the presence of non affine guards and of non affine domains. For example, the loop `for(j=1; j<m; j=2*j)` has invariant $2 \le j < m$ (in the loop) and ranking j, which gives a WCCC of m instead of the correct value $\log_2 m$. Such a WCCC cannot be obtained by an affine technique, which grossly over-estimates the domain of j by a polyhedron. Another problem is that the invariant at a loop entry is often a coarse polyhedral approximation of a union of more accurate invariants on each path in the loop. Imposing the non-negative constraint (2) for such a control point is not necessary. It is enough to impose it for one control point per circuit of the automaton where invariants are more

accurate. Note also that, if one wants to count the number of loop traversals and not the number of transitions, it is not necessary to extend the sum in (7) to all nodes. For instance, if we include only one well-chosen state per loop, we will get a bound on the total number of loop traversals in an execution of the program.

Example of Section 2.3 (Cont'd). Inequality (7) gives the upper bound:

$$\text{WCCC} \leq \#\rho(\mathcal{P}_{start}) + \#\rho(\mathcal{P}_{lbl_4}) + \#\rho(\mathcal{P}_{lbl_5}) + \#\rho(\mathcal{P}_{lbl_6}) + \#\rho(\mathcal{P}_{lbl_{10}}) + \#\rho(\mathcal{P}_{stop})$$

Let us detail the computation of $\#\rho(\mathcal{P}_{lbl_4})$. $\rho(\mathcal{P}_{lbl_4}, x, y, m) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix} (x\ y\ m)^T + \begin{bmatrix} 3 \\ 3 \end{bmatrix}$ Here, the mapping is bijective, so it would be sufficient to count the integral points in $\mathcal{P}_{lbl_4}$. But let us do the computation: $\begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \end{bmatrix} = U \times D \times V = \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 6 & 0 \end{bmatrix} \times \begin{bmatrix} -2 & 3 & 0 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.
Projecting $V\mathcal{P}_{lbl_4}$ along its first two dimensions amounts to consider the linear system: $\{p_1 = -2x+3y, p_2 = x-y, 0 < x \leq m, 0 < y \leq m\}$ and to eliminate every variable except p_1, p_2 and m (the parameter). This gives the polyhedron $\mathcal{V}$ defined by the constraints $\{1 \leq p_1 + 2p_2 \leq m, 1 \leq p_1 + 3p_2 \leq m\}$, whose number of points is an upper bound for $\#\rho(\tilde{\mathcal{P}}_{lbl_4})$. The cardinal of $\tilde{\mathcal{V}}$ is computed thanks to Ehrhart polynomials [10] (see Section 5). The result is, in general, a collection of polynomial formulas guarded by affine constraints on the parameters. Here, we get: $\#\rho(\mathcal{P}_{lbl_4}) \leq \#\mathcal{V} = m^2$ as expected. Applying the same process on the other control points, we finally obtain:

$$\text{WCCC} \leq 1 + (m^2) + (1 + 2m + m^2) + (2 + 3m + m^2) + (2m + m^2) + 1 = 5 + 7m + 4m^2.$$

5 Implementation and Experimental Results

We have built a tool suite that converts a C program into an integer interpreted automaton, constructs its invariants, tests its termination and, if successful, computes an upper bound for its worst-case computational complexity WCCC.

The first tool, c2fsm, turns a C program into an integer interpreted automaton, doing the relevant approximations when the program cannot be exactly translated. Our guidelines have been to consider only assignments to integer variables, and to give a variable an undefined value unless it is expressed as an affine form of integer variables. This tool also implements dead code elimination, useless variables elimination, and, as an option, the selection of cutpoints and the elimination of other control points. Note that it may be possible to extract flowcharts from binaries or assembly code, thus greatly extending the scope of the method.

The second tool, Aspic ([21], `http://laure.gonnord.org/aspic/`), a public-domain implementation of abstract acceleration [14], computes the invariants for every control point of the obtained integer interpreted automaton. Compared to the standard widening approach, this method computes a more precise reachability set for "accelerable" loops, which locally avoids the use of widening and globally increases precision.

The third tool, Rank, implements the method described in this paper. Starting from the integer interpreted automaton and the invariants given by Aspic, Rank tries to prove the termination of the program by computing (multidimensional affine) ranking functions. In case of success, Rank computes the worst-case computational complexity of

the program. Also, in case of failure, Rank tries to exhibit a counterexample that causes non-termination. The linear programs involved in the termination part are solved thanks to the PIP tool (Parametric Integer Programming). The WCCC part requires counting the number of points into a $\mathbb{Z}$-polyhedron. This is done thanks to the Ehrhart polynomial part of the Polylib library (`http://icps.u-strasbg.fr/polylib`). The final result is a set of Ehrhart polynomials, guarded by affine predicates on program parameters.

On the web page `http://www.ens-lyon.fr/LIP/COMPSYS/Tools/Ranking/`, a table of experimental results can be found. Examples were collected from the extent literature, and notably from `http://www.eecs.qmul.ac.uk/~aziem/esop.html`. In all test cases we were able to prove termination, even for nondeterministic examples. Nested loops are correctly handled, and we find multi-dimensional rankings for them. WCCCs are returned by Rank as piecewise functions depending on the initial values of the variables: the table only provides the most general term of these expressions.

We were also able to prove the termination of some classical sorting algorithms. The rankings for these codes may seem of the wrong dimensions, but the additional dimensions have constant values and the orders of magnitude of the WCCC are still as expected, e.g., $O(N^2)$ for `bubblesort`. For `heapsort`, our algorithm finds an $O(N^2)$ WCCC instead of the correct $O(N \log_2 N)$, see Section 4 for an explanation.

Our tools are completely autonomous within the stated limitations on input programs. The precision of the results is strongly dependent on the quality of the invariants and of the affine approximation of some (non affine) affectations in the C programs. This is not a surprise as stated by Theorem 1: the quality of our technique is to be understood with respect to the quality of invariants that are provided.

6 Related Work

Our work establishes connections with at least three different techniques. First, it brings to the field of program termination, techniques primarily designed for scheduling and optimizing do loops, in the context of automatic parallelization [17]. The fundamental difference is that, for program termination, each problem dimension corresponds to an integer variable while, for automatic parallelization, it corresponds to a predefined loop counter. In this sense, it has also some similarities with the seminal work of Karp, Miller, and Winograd on systems of uniform recurrence equations [25]. Our algorithm to generate ranking functions is inspired by the algorithm of Feautrier [19] and its completeness [32] for scheduling affine loops. Counting techniques using Ehrhart polynomials are also standard for optimizing loops [11].

Second, it extends the ranking techniques previously proposed to prove the termination of programs. Using ranking functions to prove correctness was first proposed in [20]. Early approaches were semi-automatic: one had to guess ranking functions, and then prove their correctness using some form of Hoare logic. Attempts to automate this process started, first with one-dimensional linear rankings such as in [12,28], then with multi-dimensional rankings such as in [13,8], and propositions to build some forms of polynomial rankings followed [7,15]. Unlike ours, the techniques of Podelski and Rybalchenko [28] and of Bradley, Manna, and Sipma [8] are designed for "single-path linear loops", i.e., programs abstracted by an automaton with a single node. [28] formulates the constraints to get a one-dimensional ranking if it exists using Farkas lemma,

while [8] gives a complete method to derive, for a single node, a multi-dimensional ranking. It also tries to compute the invariants and the ranking functions simultaneously. Unlike these two methods, the technique of Colón and Sipma [13] handle flowchart programs of arbitrary structure. As explained in Section 3.2, the rankings it can generate correspond to a subclass of affine rankings where all control points within the strongly connected component being considered have the same linear part. It is not complete for the class of general multi-dimensional affine ranking functions, as the examples of Section 3.2 demonstrate. Finally, none of these techniques has been designed or extended to compute upper bounds on the WCCC, i.e., the maximal length of an execution trace.

To summarize, we extend previous work on affine ranking functions in several directions. First, unlike [28,8], we are not limited to one loop, i.e., our automaton can have an arbitrary number of vertices (as in reference [13]). As shown by the example of Section 2.3, this is mandatory to analyze complex loops, either nested loops, or multi-path simple loops that have been transformed into an automaton with several vertices by path-sensitive analysis. Second, to decide at which dimension of the ranking function a transition decreases (it must be non-increasing for the previous components of the ranking), the algorithm `has_llrf` in [8, Figure 2] is a potentially-exponential recursive exploration. Since the algorithm is also potentially exhaustive, there is no need to prove completeness. In contrast, as our algorithm is greedy, a completeness proof is needed, which is also an order of magnitude more general since we deal with the much larger space of all multi-dimensional affine ranking functions, not just one single lexicographic function. Third, unlike previous papers, we are able to prove that we get the smallest number of dimensions for each ranking function. In [7], the authors do notice that they may have as many dimensions as the number of transitions. As explained in Section 4, this dimension reduction is important for the computation of the WCCC.

In a different context, a large body of research followed the introduction of the size change termination (SCT) principle in [26]. The difference in the two approaches are first in semantics: the automaton represents a call graph instead of a control graph, and the variables may be summary information about data structures, like the length of a list or the size of a tree. More importantly, the relations between input and output variables of a transition are restricted to one of the two forms $x' < y$ and $x' \leq y$. Attempts to relax this restriction can be found in [3,4,5]. Once a set of size change relations has been found, termination follows if one can combine them in such a way that one variable at least is guaranteed to decrease. Such a combination can be interpreted as a ranking function, albeit of a shape fairly different from ours. Algorithms are provided to derive rankings, with a high complexity (at least in theory) due to their combinatorial nature.

Another trend of research has been started in [29] and pursued in [9]. Here, one uses several (local) ranking relations, all of them well founded, the intuition being that each relation proves termination of a part of the program. A consistency condition is necessary: the transitive closure of the transition relation of the program must be included in the union of all local ranking relations. The problem is how to find the local rankings, and how to prove the consistency condition. It may be that we can help at least for the first problem: apply our algorithm to cleverly chosen subsets of the automaton states, as for example strongly connected components or loops. However, as pointed out in [24], how to use local rankings (instead of global ones) for WCCC computations is not clear.

The third and last connection with previous work is related to the WCCC computation. The method of Gulwani et al. [24,23] for proving termination and bounding the complexity consists in creating counters – new variables which are incremented when traversing some transitions – and asking an invariant generator for bounds on the counter values. An elaborate system is proposed for selecting the transitions to be counted, which necessitates repeated calls to the invariant generator. Our method is related to this work in the following way. After a first round of computation of ranking functions, let us create a new counter which is reset to zero at the beginning of the program and incremented at each transition satisfied by the first component of our ranking (transitions t for which the variable ε_t of Section 3.1 is equal to 1). By construction, at each control point, the sum of the counter value and the affine expression given by the ranking is non-increasing, which provides an affine bound for this counter. We can continue in this way as the construction of ranking functions progresses and transitions are removed, making sure that new counters are reset to zero at the entry to each program fragment (i.e., on incoming transitions that were previously satisfied). If and when all edges are satisfied, we have found a system of counters which meets the constraints of Gulwani et al. Hence, our approach can be seen as a replacement for the counter placement algorithm of [24]. Both techniques rely on abstract interpretation to build initial invariants. Our technique is then guaranteed to find an adequate placement of counters if one exists, given these initial invariants, while the approach of Gulwani et al. is dependent on the unavoidable approximations made in abstract interpretation to build new invariants including the counters. Which method is best from the point of view of practical complexity is difficult to ascertain, since we avoid calling the invariant generator many times, but at the price of having to solve much larger linear programming problems. However, we point out that, in [24], counters are placed only on particular transitions selected *a priori*, typically the back edges of the control-flow graph. But, in the example of Section 2.3, both back edges (the self-loop on lbl_6 and the transition from lbl_{10} to lbl_4) are traversed a quadratic number of times, so there is no transition to place a linearly-bounded counter and the algorithm of [24] would fail. As our ranking function shows, the "outer" counter should be placed either on the transition from lbl_5 to lbl_6 or on the transition from lbl_6 to lbl_{10}. Or the graph must be transformed as proposed in [23], but with a risk of complexity increase. We believe that our work bridges the gap between techniques based on the placement of counters and the use of abstract interpretation to bound them, and techniques based on *global* ranking functions to derive complexity bounds.

7 Conclusion

7.1 Contributions

The first main contribution of this paper is the design of an algorithm for the construction of multi-dimensional affine ranking functions, which, in contrast to the combinatorial algorithm of [7], is greedy but nevertheless complete (with respect to the invariants found and the class of ranking functions considered) and optimal in the dimension of the ranking function. The algorithm makes no assumption on the shape of the source

program, and can handle, with proper preprocessing (i.e., after the program is approximated to fit into the affine interpreted automaton model), multiple loops of arbitrary nesting patterns, premature termination and `goto`s, nondeterministic choices and values, exceptions, and affine guards of arbitrary structure. We also point out that, in case of failure, our algorithm may exhibit a certificate of non termination in the form of an execution trace which may not terminate. The computation of the worst-case computational complexity (WCCC) is delegated to a very comprehensive stand-alone algorithm. This means that no arbitrary restrictions about the shape of loops and tests are necessary. We can directly rely on existing methods and tools for counting integer points within $\mathbb{Z}$-polyhedra and images of $\mathbb{Z}$-polyhedra by affine functions.

More generally, our work establishes a strong link with computation models, theoretical results, and tools developed by the community of automatic parallelization and high-performance computing, which seem to be not so used (or partly re-discovered) in the context of program termination. We believe that this connection can lead to further fruitful advances in the solution of problems faced by both communities.

7.2 Future Work

There is nevertheless room for many improvements. The preprocessor we use for converting a program into an interpreted automaton is somewhat brute force: any construct that is not affine in integer variables is replaced by the bottom value, which is absorbing ($\perp \oplus x = \perp$ for most operators), and which prints as `true` in a guard and as a question mark in an action. This can be improved by noticing that some operations, like modulo and integer division, can be linearized by the introduction of fresh variables, or that a bottom value may be constrained: for instance, a square is always non-negative. Also, variables with a finite domain, like Booleans and enums, can be used to refine the states. This may result in a large increase in the size of the automaton but has the direct benefit of extending the class of ranking functions considered, as these do not need to be affine anymore for such "unrolled" variables. Making sure that domains of integer variables are "fat" (to use the terminology of [16]) increases the chance that an affine ranking exists and improves the quality of the WCCC produced.

There is always room for improving an invariant constructor like Aspic. One may for instance improve the acceleration algorithms and loops treatment, or use additional abstract interpretation frameworks, like the congruences and lattices of [22]. It may also be interesting to construct the invariants *on demand*, both to improve the accuracy and to reduce the overhead of the method.

Last but not least, the power of the ranking algorithm can be increased in many ways. For instance, imposing that ranking functions are nonnegative everywhere (see Inequality (2)) is too strong a constraint. It is enough to impose it at a set of cut points. If the automaton graph becomes acyclic when these cut points are removed, then termination is still guaranteed, notwithstanding the relaxed nonnegativity constraint. In a way, eliminating all states but cutpoints before computing a ranking (path coalescing) is equivalent to relaxing the positivity constraint, but it is obtained at the cost of a potential increase in the number of transitions: if the eliminated state has n ingoing and m outgoing transitions, its elimination will generate up to $n \times m$ transitions. We still need to explore this trade-off and analyze its consequences on the WCCC computations.

Research on the SCT paradigm has shown that ranking functions of a more complex shape, like piecewise affine functions, are necessary in some cases. In our framework, this means splitting the invariant of some state(s) by an affine constraint. How to choose the states to split and the splitting predicate is left for future research.

A point we have not investigated is the termination of distributed programs. Our algorithm fails when termination depends on a fairness hypothesis.

References

1. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Bounding the computational complexity of flowchart programs with multi-dimensional rankings. Research Report 7235, INRIA (March 2010)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L., Quinson, C.: Program termination and worst-time complexity with multi-dimensional affine ranking functions. Research Report 7037, INRIA (November 2009)
3. Anderson, H., Khoo, S.C.: Affine-based size-change termination. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 122–140. Springer, Heidelberg (2003)
4. Ben-Amram, A.M.: Size-change termination with difference constraints. ACM Transactions on Programming Languages and Systems (TOPLAS) 30(3), 1–31 (2008)
5. Ben-Amram, A.M.: Size change termination, monotonicity constraints, and ranking functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 109–123. Springer, Heidelberg (2009)
6. Blass, A., Gurevich, Y.: Inadequacy of computable loop invariants. ACM Transactions on Computational Logic (TOCL) 2(1), 1–11 (2001)
7. Bradley, A.A., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
8. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
9. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking abstractions. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 81–92. Springer, Heidelberg (2008)
10. Clauss, P.: Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In: International Conference on Supercomputing (ICS 1996), pp. 278–285. ACM, New York (1996)
11. Clauss, P.: Handling memory cache policy with integer points counting. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds.) Euro-Par 1997. LNCS, vol. 1300, pp. 285–293. Springer, Heidelberg (1997)
12. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
13. Colón, M.A., Sipma, H.B.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
14. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th ACM Symposium on Principles of Programming Languages (POPL 1978), pp. 84–96. Tucson (January 1978)
15. Cousot, P.: Proving program invariance and termination by parametric abstraction, Lagrangian relaxation, and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)

16. Darte, A., Khachiyan, L., Robert, Y.: Linear scheduling is nearly optimal. Parallel Processing Letters 1(2), 73–81 (1991)
17. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, Basel (2000) ISBN 0-8176-4149-1
18. Darte, A., Vivien, F.: Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. International Journal of Parallel Programming 25(6), 447–496 (1997)
19. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II, multidimensional time. International Journal of Parallel Programming 21(6), 389–420 (1992)
20. Floyd, R.W.: Assigning meaning to programs. In: Schwartz, J.T. (ed.) Symposium on Applied Mathematics, vol. 19, pp. 19–32. A.M.S, Providence (1967)
21. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
22. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: Abramsky, S. (ed.) TAPSOFT 1991.LNCS, vol. 494, pp. 169–192. Springer, Heidelberg (1991)
23. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 375–385. ACM, Dublin (2009)
24. Gulwani, S., Mehra, K.K., Chilimbi, T.: SPEED: Precise and efficient static estimation of program computational complexity. In: 36th ACM Symposium on Principles of Programming Languages (POPL 2009), pp. 127–139. Savannah (January 2009)
25. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. Journal of the ACM 14(3), 563–590 (1967)
26. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. ACM SIGPLAN Notices 36(3), 81–92 (2001)
27. Manna, Z.: Mathematical Theory of Computing. McGraw-Hill, New York (1974)
28. Podelski, A., Rybalchenko, A.: In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
29. Podelski, A., Rybalchenko, A.: Transition invariants. In: Ganzinger, H. (ed.) IEEE Symposium on Logic in Computer Science (LICS 2004), pp. 32–41. IEEE Computer Society, Los Alamitos (July 2004)
30. Schrijver, A.: Theory of linear and integer programming. Wiley, NewYork (1986)
31. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica 48(1), 37–66 (2007)
32. Vivien, F.: On the optimality of Feautrier's scheduling algorithm. Concurrency and Computation: Practice and Experience 15(11-12), 1047–1068 (2003); Euro-Par'02 Special Issue
33. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The determination of worst-case execution times—overview of the methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 1–53 (2008)

Deriving Numerical Abstract Domains via Principal Component Analysis

Gianluca Amato, Maurizio Parton, and Francesca Scozzari

Università di Chieti-Pescara – Dipartimento di Scienze

Abstract. We propose a new technique for developing ad-hoc numerical abstract domains by means of statistical analysis. We apply Principal Component Analysis to partial execution traces of programs, to find out a "best basis" in the vector space of program variables. This basis may be used to specialize numerical abstract domains, in order to enhance the precision of the analysis. As an example, we apply our technique to interval analysis of simple imperative programs.

1 Introduction

Numerical abstract domains are widely used to prove properties of program variables such as "all the array indexes are contained within the correct bounds" or "division by zero cannot happen". Moreover, numerical properties may help other kind of analyses, such as termination analyses [6], timing analyses [15], shape analyses [5], string cleanness analyses [10] and so on. Many numerical abstract domains strive to trade the accuracy of convex polyhedra [9] for higher speed (for instance, see the Octagon domain in [21]).

The precision of the analyses may often be improved with the use of special-purpose abstract domains, such as the domains for the analysis of digital filters [11], or the arithmetic-geometric progression abstract domain [12]. This idea may be pushed further by devising domains not just for a class of applications, but for a single program. For example, if we know the general form of the while-loop invariants which occur in a program, domains able to express these invariants should reach a higher precision than others.

In this paper we describe a family of ad-hoc domains and provide a fully automatic mechanism which, starting from an approximation of the concrete semantics of a program, selects the best domain in the family.

Consider the program in Figure 1, where the parameter x is the input and y is a local variable, and its partial execution trace for the input $x = 10$ which stops after 5 iterations of the `while` statement. Collecting the values for the variables x and y at different program points (after each assignment), we obtain the table in Figure 2. If we abstract this set of values using the box domain [7] of Cartesian product of intervals, we get the shaded area in Figure 3, given by

$$\begin{cases} 5 \leq x \leq 10 \\ -10 \leq y \leq -5 \end{cases}$$

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 134–150, 2010.
© Springer-Verlag Berlin Heidelberg 2010

```
xyline = function(x)
{
   assume(x>=0)
   y=-x
   while(x>y) {
      x= x-1
      y= y+1
   }
}
```

Fig. 1. The example program `xyline`

x	y
10	−10
9	−10
9	−9
8	−9
8	−8
7	−8
7	−7
6	−7
6	−6
5	−6
5	−5

Fig. 2. A partial execution trace of the example program

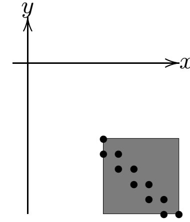


Fig. 3. Representation of the partial execution trace and relative box abstraction

The key point is that the abstraction in the box domain depends on the coordinate system we choose to draw the boxes. With the standard choice of (x, y) as coordinate system, the box in Figure 3 is a very rough approximation of the partial trace, but we can improve the precision by conveniently changing the axes. For instance, consider a different coordinate system whose axes x', y' are clockwise rotated by 30 degrees. The abstraction in this "rotated box domain" is depicted in Figure 4. The two boxes in Figure 4 are incomparable as sets of points, nonetheless the rotated box seems to fit better: for example, it has a smaller area. The question is how to find a "best rotation".

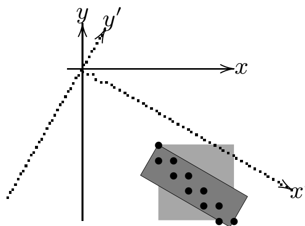


Fig. 4. Abstraction with boxes rotated by 30 degrees

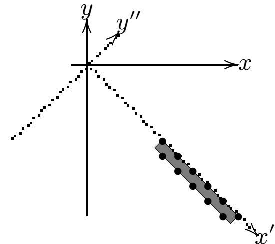


Fig. 5. Abstraction with boxes rotated by 45 degrees

To this aim, we use a statistical tool called *Principal Component Analysis* (PCA). The intuitive idea of PCA is to choose the axes maximizing the variance of the collected values. More explicitly, PCA finds a new orthonormal coordinate system such that the variance of the projection of the data points on the first axis is the maximum among all possible directions, the variance of the projection of the data points on the second axis is the maximum among all possible directions which are orthogonal to the first axis, and so on. In our example, the greatest variance is obtained by projecting the data points along the line $y = -x$. An orthogonal coordinate system (x', y') with the first axis corresponding to this

line may be obtained by a 45 degree clockwise rotation. The abstraction with respect to the box domain in (x'', y'') is depicted in Figure 5, and in the original coordinates it is given by:

$$\begin{cases} 10 \leq x - y \leq 20 \\ -1 \leq x + y \leq 0 \end{cases}$$

It is worth noting that the `while` invariant in our example is $x+y = 0, x-y \geq 0$, and it may be expressed only in the domain of 45 degree rotated boxes. This suggests that, using rotated boxes as abstract objects, an abstract interpretation based analyzer could infer this invariant. More generally, our intuition says that, if we consider well-known numerical abstract domains and adapt them to work with non-standard coordinate systems, we can improve the precision of the analysis without much degradation of performance. In this paper we develop the theoretical foundation and the implementation to validate this intuition, using the box domain as a case study. In Section 6, we will show that our analysis actually infers the invariant $x + y = 0, x - y \geq 0$.

The paper is structured as follows. Section 2 introduces some notations. Section 3 presents the abstract domains of parallelotopes, i.e. boxes w.r.t. non-standard coordinate systems, while Section 4 gives the abstract operators. Section 5 introduces PCA, used to automatically derive the "best coordinate system". Section 6 presents the prototype implementation we have developed in the R programming language [23], and shows some experimental results. Finally, in Section 8 we discuss ideas on future work.

2 Notations

Linear Algebra. We denote by $\bar{\mathbb{R}}$ the ordered field of real numbers extended with $+\infty$ and $-\infty$. Addition and multiplication are extended to $\bar{\mathbb{R}}$ in the obvious way, with the exception that 0 times $\pm\infty$ is 0. We use boldface for elements $\boldsymbol{v}$ of $\bar{\mathbb{R}}^n$. Given $\boldsymbol{u}, \boldsymbol{v} \in \bar{\mathbb{R}}^n$, and a relation $\bowtie \in \{<, >, \leq, \geq, =\}$, we write $\boldsymbol{u} \bowtie \boldsymbol{v}$ if and only if $u_i \bowtie v_i$ for each $i \in \{1, \ldots, n\}$. We denote by $\cdot$ the *dot product* on $\bar{\mathbb{R}}^n$, namely, $\boldsymbol{u} \cdot \boldsymbol{v} \stackrel{\text{def}}{=} u_1 v_1 + \cdots + u_n v_n$.

If $A = (a_{ij})$ is a matrix, we denote by A^T its *transpose*. If A is invertible, A^{-1} denotes its inverse, and $\mathrm{GL}(n)$ is the group of $n \times n$ invertible matrices. The identity matrix in $\mathrm{GL}(n)$ is denoted by I_n, and any $A \in \mathrm{GL}(n)$ such that $AA^T = I_n$ is called an *orthogonal* matrix. Clearly, any $1 \times n$-matrix can be viewed as a vector: in particular, we denote by $\boldsymbol{a}_{i*}$ (respectively $\boldsymbol{a}_{*j}$) the vector given by the i-th row (respectively the j-th column) of any $n \times n$-matrix A. If A is orthogonal, then the vectors $\boldsymbol{a}_{i*}$ are *orthonormal* (they have length 1 and are orthogonal), and thus are linearly independent. The same holds for vectors $\boldsymbol{a}_{*j}$. The standard orthonormal basis of $\mathbb{R}^n$ is denoted by $\{\boldsymbol{e}^1, \ldots, \boldsymbol{e}^n\}$.

Abstract Interpretation. (See [8] for details). Given complete lattices $(C, \leq_C)$ and $(A, \leq_A)$, respectively called the *concrete domain* and the *abstract domain*,

a *Galois connection* is a pair (α, γ) of monotone maps $\alpha : C \to A$, $\gamma : A \to C$ such that $\alpha\gamma \leq_A \mathrm{id}_A$ and $\gamma\alpha \geq_C \mathrm{id}_C$. If $\alpha\gamma = \mathrm{id}_A$, then (α, γ) is called a *Galois insertion*. Given a monotone map $f : C \to C$, the map $\tilde{f} : A \to A$ is a *correct approximation* of f if $\alpha f \leq \tilde{f}\alpha$. The *best correct approximation* of f is the smallest correct approximation f^α of f. It is well-known that $f^\alpha = \alpha f \gamma$.

Boxes. A set $\mathcal{B} \subseteq \mathbb{R}^n$ is called a (closed) *box* if there are *bounds* $\boldsymbol{m}, \boldsymbol{M} \in \bar{\mathbb{R}}^n$ such that

$$\mathcal{B} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{m} \leq \boldsymbol{x} \leq \boldsymbol{M}\} \ .$$

We denote such a box with $\langle \boldsymbol{m}, \boldsymbol{M} \rangle$. Boxes are used to abstract subsets of $\mathbb{R}^n$. If $\mathbb{B}\mathsf{ox}$ is the set of all the boxes, a Galois insertion $(\alpha^{\mathbb{B}}, \gamma^{\mathbb{B}}) : \wp(\mathbb{R}^n) \rightleftharpoons \mathbb{B}\mathsf{ox}$ may be defined by letting $\alpha^{\mathbb{B}}(\mathcal{C})$ be the smallest box enclosing $\mathcal{C}$ and $\gamma^{\mathbb{B}}(\mathcal{B}) = \mathcal{B}$.

Given two boxes $\langle \boldsymbol{m}, \boldsymbol{M} \rangle, \langle \boldsymbol{m}', \boldsymbol{M}' \rangle \in \mathbb{B}\mathsf{ox}$, we will use the following notation for the standard box operations (see [7]):

- The abstract union operation $\langle \boldsymbol{m}, \boldsymbol{M} \rangle \cup^{\mathbb{B}} \langle \boldsymbol{m}', \boldsymbol{M}' \rangle$ yields the smallest box containing both $\langle \boldsymbol{m}, \boldsymbol{M} \rangle$ and $\langle \boldsymbol{m}', \boldsymbol{M}' \rangle$;
- The abstract intersection operation $\langle \boldsymbol{m}, \boldsymbol{M} \rangle \cap^{\mathbb{B}} \langle \boldsymbol{m}', \boldsymbol{M}' \rangle$ computes the greatest box contained in both $\langle \boldsymbol{m}, \boldsymbol{M} \rangle$ and $\langle \boldsymbol{m}', \boldsymbol{M}' \rangle$;
- $\mathsf{assign}^{\mathbb{B}}(i, \boldsymbol{a}, b) : \mathbb{B}\mathsf{ox} \to \mathbb{B}\mathsf{ox}$ corresponds to the (linear) assignment "$x_i = \boldsymbol{a} \cdot \boldsymbol{x} + b$", where $\boldsymbol{x}$ is a vector of n variables, $i \in \{1, \ldots n\}$, $\boldsymbol{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. It is the best correct abstraction of the concrete operation $\mathsf{assign}(i, \boldsymbol{a}, b) : \wp(\mathbb{R}^n) \to \wp(\mathbb{R}^n)$ defined as the pointwise extension of:

$$\mathsf{assign}(i, \boldsymbol{a}, b)(\boldsymbol{x}) \stackrel{\text{def}}{=} \boldsymbol{y} \quad \text{where} \quad y_j = \begin{cases} x_j & \text{if } j \neq i \ , \\ (\boldsymbol{a} \cdot \boldsymbol{x}) + b & \text{if } j = i \ . \end{cases}$$

- $\mathsf{test}^{\mathbb{B}}(\boldsymbol{a}, b, \bowtie) : \mathbb{B}\mathsf{ox} \to \mathbb{B}\mathsf{ox}$ corresponds to the *then*-branch of the if-statement "`if` $(\boldsymbol{a} \cdot \boldsymbol{x} \bowtie b)$", where $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$. It is the best correct abstraction of the concrete operation $\mathsf{test}(\boldsymbol{a}, b, \bowtie) : \wp(\mathbb{R}^n) \to \wp(\mathbb{R}^n)$ defined as:

$$\mathsf{test}(\boldsymbol{a}, b, \bowtie)(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{C} \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \cdot \boldsymbol{x} \bowtie b\} \ .$$

  The abstract operation $\mathsf{test}^{\mathbb{B}}(\boldsymbol{a}, b, \bowtie)(\langle \boldsymbol{m}, \boldsymbol{M} \rangle)$ computes the smallest box which contains the intersection of $\langle \boldsymbol{m}, \boldsymbol{M} \rangle$ and the set of points $\{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \cdot \boldsymbol{x} \bowtie b\}$.

3 The Parallelotope Domains

Every choice of $A \in \mathrm{GL}(n)$ gives new coordinates in $\mathbb{R}^n$, and boxes with respect to this transformed coordinates are called *parallelotopes*. Thus, a parallelotope is a box whose edges are parallel to the axes in the new coordinate system. Remark that we are not restricting to orthogonal change of basis. This means that we consider any invertible linear transformation, such as rotation, reflection, stretching, compression, shear or any combination of these. The aim of the change of coordinate system is to fit the original data with a higher precision than with standard boxes.

Example 1. Consider the set $\mathcal{C} = \{(u, -u) \mid u \geq 0\} \subseteq \mathbb{R}^2$ corresponding to the `while` invariant $x+y = 0, x-y \geq 0$ of program in Figure 1. If we directly abstract $\mathcal{C}$ in the box domain, we get $\alpha^{\mathbb{B}}(\mathcal{C}) = \langle(0, -\infty), (+\infty, 0)\rangle$, and $\gamma^{\mathbb{B}}(\alpha^{\mathbb{B}}(C)) = \mathbb{R}^+ \times \mathbb{R}^-$, with a sensible loss of precision. Let us consider a clockwise rotation of 45 degrees, centered on the origin, of the standard coordinate system. The matrix

$$A = \begin{bmatrix} \cos(-\frac{\pi}{4}) & -\sin(-\frac{\pi}{4}) \\ \sin(-\frac{\pi}{4}) & \cos(-\frac{\pi}{4}) \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

transforms rotated coordinates into standard coordinates.

We want to abstract $\mathcal{C}$ with boxes on the rotated coordinate system. To this aim, we first compute the rotated coordinates of the points in $\mathcal{C}$, and then compute the smallest enclosing box. Since the rotated coordinates are given by $A^{-1}(x, y)^T$, we obtain:

$$\begin{aligned} \alpha^{\mathbb{B}}(A^{-1}\mathcal{C}) &= \alpha^{\mathbb{B}}(\{A^{-1}\boldsymbol{v} \mid \boldsymbol{v} \in \mathcal{C}\}) \\ &= \alpha^{\mathbb{B}}\left(\left\{\begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} u \\ -u \end{bmatrix} \mid u \in \mathbb{R}^+\right\}\right) \\ &= \alpha^{\mathbb{B}}\left(\left\{\begin{bmatrix} u\sqrt{2} \\ 0 \end{bmatrix} \mid u \in \mathbb{R}^+\right\}\right) = \langle(0, 0), (+\infty, 0)\rangle \ . \end{aligned}$$

The axes in the rotated coordinate system are, respectively, the lines $y = -x$ and $y = x$ in the standard coordinate system. It means that the box $\langle(0, 0), (+\infty, 0)\rangle$ computed above may be represented algebraically in the standard coordinate system as

$$\begin{cases} 0 \leq x + y \leq 0 \\ 0 \leq x - y \leq +\infty \end{cases}$$

More in general, using the matrix A, we may represent all the parallelotopes of the form

$$\begin{cases} m_1 \leq x + y \leq M_1 \\ m_2 \leq x - y \leq M_2 \end{cases}$$

Thus, we have transformed a non-relational analysis into a relational one, where the form of the relationships is given by the matrix A. If we concretize the box by applying $\gamma^{\mathbb{B}}$ and using the matrix A to convert the result to the standard coordinate system, we obtain $A\gamma^{\mathbb{B}}\alpha^{\mathbb{B}}(A^{-1}\mathcal{C}) = \mathcal{C}$. Thus, we get a much better precision than using standard boxes. We stress out that we need to choose A cleverly, on the base of the specific data set, otherwise we may loose precision: for example, if $\mathcal{D} = \{(u, 0) \mid u \in \mathbb{R}\}$, then $\gamma^{\mathbb{B}}(\alpha^{\mathbb{B}}(\mathcal{D})) = \mathcal{D}$ but $A\gamma^{\mathbb{B}}\alpha^{\mathbb{B}}(A^{-1}\mathcal{D}) = \mathbb{R}^2$.

It is worth noting that, if we prefer not to deal with irrational numbers, we may choose the transformation matrix

$$A' = \sqrt{2}\ A = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

This corresponds to a 45 degree clockwise rotation followed by a scaling by $\sqrt{2}$ in all directions.

In order to define the abstract domains of parallelotopes, we use the same complete lattice $\mathbb{B}\mathsf{ox}$ we used for the box domain, but equipped with a different abstraction function, and different abstract operations.

Definition 1 (The Parallelotope Domains). *Given $A \in \mathrm{GL}(n)$, we define the maps $\gamma^A : \mathbb{B}\mathsf{ox} \to \wp(\mathbb{R}^n)$ and $\alpha^A : \wp(\mathbb{R}^n) \to \mathbb{B}\mathsf{ox}$ as*

$$\gamma^A(\langle \boldsymbol{m}, \boldsymbol{M} \rangle) \stackrel{\text{def}}{=} A\gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle) \ ,$$

$$\alpha^A(\mathcal{C}) \stackrel{\text{def}}{=} \alpha^{\mathbb{B}}(A^{-1}\mathcal{C}) \ .$$

Using the above definition, it is easy to check that (α^A, γ^A) is a Galois insertion. Intuitively, the abstraction α^A first projects the points into the new coordinate system, then computes the standard box abstraction. The concretization map γ^A performs the opposite process. Remark that, as a particular case, we have $\alpha^{I_n} = \alpha^{\mathbb{B}}$ and $\gamma^{I_n} = \gamma^{\mathbb{B}}$.

4 Abstract Operations on Parallelotopes

In this section we illustrate the main abstract operations on the Parallelotope domains. We show that, in most cases, the abstract operators can be easily recovered by the corresponding operators on boxes. In all the operations, we ignore the computational cost of computing the inverse of the matrix A. If A is orthogonal, the cost may be considered constant since $A^{-1} = A^T$ and we do not need to compute the transpose: it is enough to consider specific algorithms which performs transposition "on the fly" when needed. If A is not orthogonal, the inverse may be computed with standard algorithms which have complexities between quadratic and cubic. However, A^{-1} needs to be computed only once for the entire execution of the abstract interpretation procedure, hence its computational cost is much less relevant then the cost of the abstract operations.

4.1 Union and Intersection

Given $B_1, B_2 \in \mathbb{B}\mathsf{ox}$, the best correct approximation of the concrete union is:

$$B_1 \cup^A B_2 \stackrel{\text{def}}{=} \alpha^A(\gamma^A(B_1) \cup \gamma^A(B_2)) \ .$$

By replacing α^A and γ^A with their definitions, A and A^{-1} cancel out and we have that $\cup^A$ is the same as $\cup^{\mathbb{B}}$. The same holds for intersection.

Proposition 1 (Union and intersection). *Given $B_1, B_2 \in \mathbb{B}\mathsf{ox}$, we have that:*

$$B_1 \cup^A B_2 = B_1 \cup^{\mathbb{B}} B_2 \qquad\qquad B_1 \cap^A B_2 = B_1 \cap^{\mathbb{B}} B_2$$

The computational complexity of both operations is $O(n)$.

4.2 Assignment

The abstract operation $\mathsf{assign}^A(i, \boldsymbol{a}, b)$ corresponds to the (linear) assignment "$x_i = \boldsymbol{a}{\cdot}\boldsymbol{x}{+}b$", where $\boldsymbol{x}$ is a vector of n variables, $i \in \{1, \dots n\}$, $\boldsymbol{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. We look for a constructive characterization of the best correct approximation, defined as:

$$\mathsf{assign}^A(i, \boldsymbol{a}, b) \stackrel{\text{def}}{=} \alpha^A \, \mathsf{assign}(i, \boldsymbol{a}, b) \, \gamma^A \ .$$

Let us note that the concrete operation may be rewritten using matrix algebra as $\mathsf{assign}(i, \boldsymbol{a}, b)(\boldsymbol{x}) = Z_{i,\boldsymbol{a}}\boldsymbol{x} + b\boldsymbol{e}^i$ where

$$Z_{i,\boldsymbol{a}} = I + \boldsymbol{e}^i \cdot \boldsymbol{a}' \quad \text{with} \quad a'_j = \begin{cases} a_j & \text{if } j \neq i, \\ a_i - 1 & \text{if } j = i. \end{cases}$$

This allows us to prove the following:

Theorem 1 (Assignment). *Given $\langle \boldsymbol{m}, \boldsymbol{M} \rangle \in \mathbb{B}\mathsf{ox}$, we have that*

$$\mathsf{assign}^A(i, \boldsymbol{a}, b)(\langle \boldsymbol{m}, \boldsymbol{M} \rangle) = \langle \boldsymbol{m}' + A^{-1}b\boldsymbol{e}^i, \boldsymbol{M}' + A^{-1}b\boldsymbol{e}^i \rangle$$

where

$$\boldsymbol{m}' = \inf_{\boldsymbol{x} \in \langle \boldsymbol{m}, \boldsymbol{M} \rangle} (H^T \boldsymbol{e}^i) \cdot \boldsymbol{x} \qquad \boldsymbol{M}' = \sup_{\boldsymbol{x} \in \langle \boldsymbol{m}, \boldsymbol{M} \rangle} (H^T \boldsymbol{e}^i) \cdot \boldsymbol{x} \ ,$$

and $H = A^{-1} Z_{i,\boldsymbol{a}} A$. The complexity is $O(n^2)$.

Proof (Sketch). We may rewrite the abstract operator as:

$$\begin{aligned} & \alpha^A(\mathsf{assign}(i, \boldsymbol{a}, b)(\gamma^A(\langle \boldsymbol{m}, \boldsymbol{M} \rangle))) \\ = \ & \alpha^{\mathbb{B}}(A^{-1}\mathsf{assign}(i, \boldsymbol{a}, b)(A\gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle))) \\ = \ & \alpha^{\mathbb{B}}(A^{-1} Z_{i,\boldsymbol{a}} A \ \langle \boldsymbol{m}, \boldsymbol{M} \rangle + A^{-1} b \boldsymbol{e}^i) \ . \end{aligned}$$

Let $H = A^{-1} Z_{i,\boldsymbol{a}} A$ and $H \ \langle \boldsymbol{m}, \boldsymbol{M} \rangle = \langle \boldsymbol{m}', \boldsymbol{M}' \rangle$. For each $i \in \{1, \dots, n\}$, $m'_i = \inf_{\boldsymbol{x} \in \langle \boldsymbol{m}, \boldsymbol{M} \rangle} (H\boldsymbol{x}) \cdot \boldsymbol{e}^i = \inf_{\boldsymbol{x} \in \langle \boldsymbol{m}, \boldsymbol{M} \rangle} (H^T \boldsymbol{e}^i) \cdot \boldsymbol{x}$. Since $H^T \boldsymbol{e}^i$ is the transpose of the i-th row of H, we may compute $\inf_{\boldsymbol{x} \in \langle \boldsymbol{m}, \boldsymbol{M} \rangle} (H^T \boldsymbol{e}^i) \cdot \boldsymbol{x}$ using interval arithmetic.

4.3 Test

We want to find a constructive characterization of the best correct approximation $\mathsf{test}^A(\boldsymbol{a}, b, \leq)$ defined as:

$$\mathsf{test}^A(\boldsymbol{a}, b, \leq) \stackrel{\text{def}}{=} \alpha^A \, \mathsf{test}(\boldsymbol{a}, b, \leq) \, \gamma^A \ .$$

Given $\langle \boldsymbol{m}, \boldsymbol{M} \rangle \in \mathbb{B}\mathsf{ox}$, we have that

$$\begin{aligned} & \alpha^A(\mathsf{test}(\boldsymbol{a}, b, \leq)(\gamma^A(\langle \boldsymbol{m}, \boldsymbol{M} \rangle))) \\ = \ & \alpha^{\mathbb{B}}(A^{-1}\mathsf{test}(\boldsymbol{a}, b, \leq)(A \ \gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle))) \\ = \ & \alpha^{\mathbb{B}}(A^{-1}((A \ \gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle)) \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \cdot \boldsymbol{x} \leq b\})) \\ = \ & \alpha^{\mathbb{B}}(\gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle) \cap \{A^{-1}\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \cdot \boldsymbol{x} \leq b\}) \\ = \ & \alpha^{\mathbb{B}}(\gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle) \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid (A^T \boldsymbol{a}) \cdot \boldsymbol{x} \leq b\}) \\ = \ & \alpha^{\mathbb{B}}(\mathsf{test}(\boldsymbol{a}, b, \leq)(\gamma^{\mathbb{B}}(\langle \boldsymbol{m}, \boldsymbol{M} \rangle))) \ . \end{aligned}$$

Hence, the abstract operator $\mathsf{test}^A(\boldsymbol{a}, b, \leq)$ can be easily computed by using the standard abstract operator on boxes, as $\mathsf{test}^A(\boldsymbol{a}, b, \leq) = \mathsf{test}^{\mathbb{B}}(A^T\boldsymbol{a}, b, \leq)$.

Proposition 2 (Test). *We have that*

$$\mathsf{test}^A(\boldsymbol{a}, b, \leq) = \mathsf{test}^{\mathbb{B}}(A^T\boldsymbol{a}, b, \leq) \ .$$

The computational complexity is $O(n^2)$.

The complexity of the algorithm to compute $\mathsf{test}^{\mathbb{B}}(\boldsymbol{a}, b, \leq)$ is $O(n)$. Thus, the complexity of computing $\mathsf{test}^A(\boldsymbol{a}, b, \leq)$ is $O(n^2)$, since we need to add the complexity for computing $A^T\boldsymbol{a}$. However, the latter might be computed only once in the analysis, and then memorized, in order to be reused every time we find such a conditional.

It is easy to see that, in the general case, the abstract counterpart of the operation $\mathsf{test}(\boldsymbol{a}, b, \bowtie)$ corresponding to the *then*-branch of the if-statement "`if` $(\boldsymbol{a} \cdot \boldsymbol{x} \bowtie b)$", where $\boldsymbol{a} \in \mathbb{R}^n$, $b \in \mathbb{R}$ and $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$, can be easily recovered by the corresponding operator on boxes. The same holds for the *else*-branch of the if-statement. For instance, the *else*-branch of "`if` $(\boldsymbol{a} \cdot \boldsymbol{x} \leq b)$" is exactly the *then*-branch of "`if` $(\boldsymbol{a} \cdot \boldsymbol{x} > b)$".

4.4 On the Implementation of Abstract Operators

Correctness of the abstract operators in actual implementations strictly depends on the exactness of matrix operations. The easiest way to ensure correctness is to use rational arithmetic. Alternatively, we could estimate the error of the floating point implementations of these operations, and use rounding to correctly approximate the abstract operators on real numbers, following the approach in [20].

Note that, although we have presented our domain as an abstraction of $\wp(\mathbb{R}^n)$, we may apply the same construction to build an abstraction of $\wp(\mathbb{Z}^n)$, in order to analyze programs with integer variables. In this case, whenever A is an integer matrix, we may perform almost all the computations on integers. In fact, observe that A^{-1} is an integer matrix divided by an integer number $d \in \mathbb{Z}$. Since all the operations involved in assign^A are linear, d may be factored out and only applied at the end, before rounding the intervals to integer bounds.

5 Principal Component Analysis

Principal component analysis (PCA) is a standard technique in statistical analysis which transforms a number of possibly correlated variables into uncorrelated variables called *principal components*, ordered by the most to the least important. Consider an $n \times m$ data matrix D on the field of real numbers. Each row may be though of as a different instance of a statistical population, while columns are attributes (see, for instance, the 11×2 matrix in Figure 2). Consider a vector $\boldsymbol{v} \in \mathbb{R}^m$, which expresses a linear combination of the attributes of

the population. The projection of the n rows of the matrix D onto the vector $\boldsymbol{v}$ is given by $D\boldsymbol{v}$. Among the different choices of $\boldsymbol{v}$, we are interested in the ones which maximize the (sample) *variance* of $D\boldsymbol{v}$. We recall that the variance of a vector $\boldsymbol{u} \in \mathbb{R}^m$ is $\sigma^2_{\boldsymbol{u}} = \frac{1}{m}\sum_{i=1}^{m}(u_i - \bar{u})^2$ where $\bar{\boldsymbol{u}}$ is the (empirical) *mean* of $\boldsymbol{u}$, i.e. $\bar{\boldsymbol{u}} = \frac{1}{m}\sum_{i=1}^{m} u_i$. Any unit vector which maximizes the variance may be chosen as the first principal component. This represents the axis which best explains the variability of data. The search for the second principal component is similar, looking for vectors $\boldsymbol{v'}$ which are orthonormal to the first principal component and maximize the variance of $D\boldsymbol{v'}$. In turn, the third principal component should be orthonormal to the first twos, with maximal variance, and so on.

From a mathematical point of view, principal component analysis finds an orthogonal matrix that transforms the data to a new coordinate system. The columns (called principal components) are ordered according to the variability of the data that they are able to express. It turns out that the columns are the eigenvectors of the *covariance matrix* of D, i.e. an $n \times n$ symmetric matrix Q such that q_{ij} is the (sample) *covariance* of d_{i*} and d_{j*}. We recall that the covariance of two vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^m$ is $\sigma_{\boldsymbol{vw}} = \frac{1}{m}\sum_{i=1}^{m}(v_i - \bar{\boldsymbol{v}})(w_i - \bar{\boldsymbol{w}})$. The columns are ordered according to the corresponding eigenvalues. However, principal components are generally computed using singular value decomposition for a greater accuracy.

Example 2. Consider the partial execution trace in Figure 2 as data matrix D. If we perform the PCA on D, we get the principal components $(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$ and $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, corresponding to the change of basis matrix

$$A = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

given in Example 1.

5.1 Orthogonal Simple Component Analysis

It is worth noting that small changes in the data cause small changes in the principal components which, however, may cause a big loss in precision. This depends on the interactions between the PCA and the parallelotope abstraction function: If $\mathcal{D}$ is an unbounded set of points (in $\mathbb{R}^n$), the bounds (in $\bar{\mathbb{R}}$) of the minimum enclosing box of $\mathcal{D}$ are not continuous w.r.t. the change of basis matrix.

Example 3. We consider the 10×2 matrix obtained removing the last line from the table in Figure 2. If we perform the PCA, we get the change of basis matrix

$$S = \begin{bmatrix} s_1 & s_2 \\ -s_2 & s_1 \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{1}{2} + \frac{1}{2\sqrt{257}}} & \sqrt{\frac{1}{2} - \frac{1}{2\sqrt{257}}} \\ -\sqrt{\frac{1}{2} - \frac{1}{2\sqrt{257}}} & \sqrt{\frac{1}{2} + \frac{1}{2\sqrt{257}}} \end{bmatrix}$$

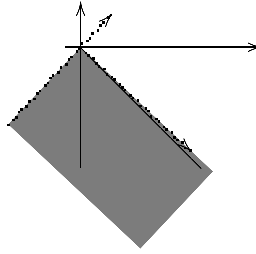


Fig. 6. Bad precision with PCA

corresponding to a clockwise rotation of about 43 degrees. The principal components are not very different from the previous ones, but now we are not able to represent parallelotopes bounded by constraints on $x+y$ and $x-y$. Therefore, the invariant $x+y=0, x-y \geq 0$ cannot be represented directly. Since the difference between the first principal component and the axis $x+y=0$ is unbounded, it is abstracted into $-\infty \leq s_2 x + s_1 y \leq 0$ and $0 \leq s_1 x - s_2 y \leq +\infty$, which is the shaded area in Figure 6. This cause a serious loss of accuracy.

In order to overcome the difficulties outlined above, we need a way of "stabilizing" the result of the PCA, so that it is less sensible to small changes in the data. There are two possible approaches to this problem: we may remove outliers (i.e., points which are very "different" from the others) by the execution trace before computing the PCA, or refine the result of the PCA. In this paper, we follow the second approach, since we prefer to maintain the whole set of original data. Our idea is that, in many cases, we expect that optimal parallelotopes which abstracts program states should contain only linear constraints with integer coefficients. This is obvious for programs with integer variables only (such as Example 1), or when we are interested in properties described by integer values (such as bounds of arrays, division by 0, etc. . .). Therefore, we would like to minimally change the result of the PCA (the matrix A) in such a way that A^{-1} is an integer matrix. Of course, the new matrix is not going to be the matrix with principal components anymore, but in this way we compensate for possible deviations of the principal components with respect to the "optimal" vectors.

There are several procedures in the statistic literature for simplifying the result of the PCA, in order to obtain integer matrices. In this paper we follow the approach of the *orthogonal simple component analysis*, introduced in [1]. The authors define a simplification procedure which transforms the result A of the PCA in an integer matrix B such that the columns of B are orthogonal and the angle between any column of A and the corresponding column of B is less then a specified threshold θ. Note that, in the general case, matrix B is not orthogonal because the columns of B are orthogonal but not orthonormal (i.e., their length is not one). This is not a problem since our domains of parallelotopes do not require matrices to be orthogonal. Note that, although B^{-1} may contain non-integer elements, each row is exactly an integer vector multiplied by a rational. Hence, it expresses integer constraints.

6 Implementation

In order to investigate on the feasibility of the ideas introduced above, we have developed a prototypical implementation for the intra-procedural analyses of a simple imperative language. The analyses may be performed with either the standard domain of boxes, the domains of parallelotopes, or with their combination. In order to collect the partial execution traces, the program under analysis is automatically augmented for recording the values of the variables at every program point. The implementation automatically recovers partial execution traces starting from the input values (which may be randomly generated or provided by the user), computes the orthogonal simple components, and performs static analysis with the three domains. Program equations are solved with a *recursive chaotic iteration strategy* on the *weak topological ordering* induced by the program structure (see [4]). The analyzer uses the standard widening [9] which extrapolates unstable bounds to infinity and the standard narrowing which improves infinite bounds only.

The prototype has been written in R [23], a language and environment for statistical computing. R is a functional language with call-by-value semantics, powerful meta-programming features, vectors as primitive data types and a huge library of built-in statistical functions. The benefits we got using R for developing our application were many. For example, thanks to the powerful meta-programming features, it was easy to augment programs with instructions which record the partial execution traces, and there was no need to implement a parser for the static analyzer. Actually, code can be manipulated programmatically in R, as in Lisp. Moreover, the vast library of statistical functions allowed us to implement easily the PCA (just a function call was sufficient) and the simplification procedure for obtaining the orthogonal simple components. Correctness of abstract operators was ensured using rational arithmetic.

The main drawback of R, at least for our application, is speed. Since it only supports call-by-value semantics, manipulating complex data structures may require several internal copy operations. For a prototype, this was deemed less important then fast coding. However, this means that we cannot compare the effective speed of the Parallelotope domains with the speed of octagons or polyhedra, because all the standard implementations of the latter domains, in libraries such as `APRON` [18] or `PPL` [2], are in C or C++.

6.1 Optimizing the Parallelotope Domains

Using the Parallelotope domains, we have occasionally experimented some problems in the bootstrap phase of the analysis. Consider the sample program `start1` in Figure 7. If we perform the analyses with the standard box domain, we may easily infer that, at the end of the function, both the variables x and y assume the value 10. However, using the Parallelotope domain with the axes clockwise rotated by 45 degrees, the analysis starts with the abstract state which covers the entire $\mathbb{R}^2$. The assignment $x = 10$ has no effect: since there are no bounds on the possible values for y, then nothing may be said about $x + y$ and $x - y$, even if we

```
start1 = function()
{
  x=10
  y=x
}
```

```
start2 = function(x)
{
  y=10
  x=y
}
```

```
cousot78 = function()
{
  i=2
  j=0
  while (TRUE) {
    if (i*i==4)
      i=i+4
    else {
      j=j+1
      i=i+2
} } }
```

Fig. 7. Example programs

know the value of x. Therefore, after the second assignment, we only know that $x - y = 0$, loosing precision with respect to the standard box analysis, although $x = y = 10$ may be expressed in the rotated domain as $x + y = 20, x - y = 0$.

The problem arises from the fact that assignments are naturally biased towards the standard axes, since the left hand side is always a variable. At the beginning of the analysis, when the abstract state does not contain any constraint, all constant assignments are lost, and this is generally unfavorable to the precision of the analysis. For this reason, our analyzer initializes all the local variables to zero, as done in many programming languages. Unfortunately, this does not always solve the problem, due to the presence of input parameters. Consider the program `start2` in Figure 7. In this case, we assume that $y = 0$ at the beginning of the function, but we cannot assume that $x = 0$, since this is a parameter. However, our parallelotope (the 45 degree clockwise rotated boxes) cannot express the fact that $y = 0$. Hence the abstract state at the beginning of the function is the full space $\mathbb{R}^2$, and the result at the end of the function is again $x - y = 0$. From the point of view of precision, an optimal solution to this kind of problems would be to use the reduced product of the box domain and Parallelotope domains. However, this may severely degrade performance. A good trade-off could be to perform both analysis in parallel: at the end of each abstract operations, we use the information which comes from one of the two domains to refine the other, and vice versa. Given a box and a parallelotope, a satisfactory and computationally affordable solution is to compute the smallest parallelotope which contains the box, and then the intersection between the two parallelotopes. The symmetric process can be used to refine the box. We have adopted this solution in our analyzer (see [11,12] for a similar approach).

6.2 Experimental Evaluation

We applied the analyzer to different toy programs we collected from the literature. Although an exhaustive comparison of the speed and precision of the domains of parallelotopes with other domains is outside the scope of this paper, we present here some preliminary results. We considered the following programs: `bsearch`: binary search over 100-element arrays, as appeared in [7]; `xyline`: the

program	Box	Parallelotope	combined	Octagon
bsearch	$1 \leq lwb \leq 100$ $1 \leq upb \leq 100$ $0 \leq m \leq 100$ $(-99 \leq upb - lwb)$ $(-100 \leq m - lwb)$	$0 \leq upb - lwb$	as box+ptope	as box+ptope+ $-99 \leq m - lwb$
bsearch*	as above	$0 \leq upb - lwb$ $-101 \leq -upb - lwb + 2m \leq 50.5$ $(-50.5 \leq m - lwb \leq 74.75)$	as box+ptope	as above
xyline		$-x + y \leq 0$ $x + y = 0$	as ptope	as ptope
bsort	$1 \leq b \leq +\infty$ $0 \leq j \leq +\infty$ $0 \leq t \leq +\infty$		as box	$1 \leq b \leq 100$ $0 \leq j \leq 100$ $0 \leq t \leq 99$ $b + j \leq 199$ $b + t \leq 198$ $0 \leq j - t$ $0 \leq b - t$
bsort*	as above	$1 \leq b$	$1 \leq b \leq 100$ $0 \leq j \leq 100,$ $0 \leq t \leq 99$ $0 \leq j - t$	as above
cousot78	$2 \leq i$ $0 \leq j$	$2 \leq i + j$ $-i + j \leq -2$	as box+ptope	as box+ptope
cousot78†	as above	$-\infty \leq -i + 2j \leq -2$	as box+ptope	as box+ptope

Fig. 8. Results of the analyses for several programs and domains. Constraints in parentheses are not part of the result of the analyses, but may be inferred from them.

example program in Figure 1; `bsort`: bubblesort over 100-element arrays, which is the first example program in [9]; `cousot78`: the program in Figure 7, which is an instance of a skeletal program in [9].

All programs have at least one loop. For each program, we show the abstract state inferred by the analyzer at the beginning of the loop. Since `bsort` has two nested loops, we only show the abstract state for the outer one. In order to compare our results to the Octagon domain [21], we have used the Interproc analyzer [17,18], enabling the option for guided analysis (see [13]). This has required converting the sample programs from the R syntax to the syntax supported by Interproc. For the parallelotope and combined domains, we have used a change of basis matrix determined by orthogonal simple component analysis with an accuracy threshold of $\cos \frac{\pi}{4}$. The only exception is `cousot78`†, where we have used an accuracy of 0.98. For `bsort` and `bsearch` we have shown two different results: the first one is for a standard analyses, while in the second one we have instructed the tracer and analyzer not to consider the variables `k` and `tmp` respectively. The variable `k` is the key for the binary search, while `tmp` is just a temporary variable used to swap two elements of an array. Both are either compared with array elements or assigned to/from array elements, but our analyzer does not deal with arrays at all, nor does Interproc. Removing these variables by the analysis helps the PCA procedure. This suggests a possible improvement, not implemented yet, which is to automatically remove from the partial execution traces those variables which are assigned to/from array elements, or compared with them.

The results show that, in most cases, the domains of parallelotopes gives interesting properties, which cannot be inferred by the corresponding results of the box domain. In the `bsort*` case, the domain of parallelotopes does not yield anything interesting, but its combination with standard boxes does: the combined domains is able to prove (like Octagon) that all accesses to arrays are correct. In most of the cases, Octagon was able to obtain more precise abstract states than ours, but the theoretical complexity of its operations is greater. However, in the `bsearch*` case we were able to obtain a property which cannot be represented in Octagon, and cannot be inferred by the corresponding results. A practical comparison of speed is not possible at the moment, since our implementation in R is definitively slower than the APRON [18] library used in Interproc.

7 Related Work

The idea of parametrizing analyses for a single program, or a class of programs, has been pursued in a few papers. The analysis for digital filters proposed in [11] is an example of domains developed for a specific class of applications. The same holds for the domain of arithmetic-geometric progressions [12], used to determine restrictions on the value of variables, as a function of the program execution time.

In our paper, we extend this idea and propose parametric domains which may be specialized for a single program. The same approach can be found in the domain of symbolic intervals [24], which depend on a total ordering of variables in the program, and most importantly, in the domain of template polyhedra [25], that is, domains of fixed form polyhedra. For each program, the authors fix a matrix A and consider all the polyhedra of type $A\boldsymbol{x} \leq \boldsymbol{b}$. The choice of the matrix is what differentiates template polyhedra from other domains, where the matrix is fixed for all programs (such as intervals or Octagon) or varies freely (such as polyhedra.)

However, in all these papers, the choice of the parameters is performed using a syntactic inspection of the program. To the best of our knowledge, the present work is the first attempt of inferring parameters on the base of partial execution traces. Moreover, we try to be as conservative as possible, and reuse the operators of the original abstract domains, instead of devising completely new operators.

There are also parametrization strategies applicable to almost all numeric domains. For example, the accuracy of widening operators can be enhanced through the adoption of intermediate thresholds [3], from a simple syntactic analysis of the program (e.g., maximum size of arrays, constants declared in the program). Moreover, the complexity of relational analyses can be reduced by using *packing*, which partitions the set of all program variables into groups, performs relational analyses within the partitions and non-relational analyses between the partitions [3]. These strategies are orthogonal to our approach, and can be applied to our domains as well.

A different approach which exploits execution traces can be found in [16]. The authors collect (probabilistic) execution traces, in order to directly derive linear relationships between program variables, which hold with a given probability. On the contrary, in our approach we use the information gathered from partial execution traces as an input for a subsequent static analysis.

8 Conclusions and Future Work

We have presented a new technique for shaping numerical abstract domains to single programs, by applying a "best" linear transformation to the space of variable values. One of the main advantages of this technique is the ability to transform non-relational analysis into relational ones, by choosing the abstract domain which best fits for a single program. Moreover, this idea may be immediately applied to any numerical abstract domain which is not closed by linear transformations, such as octagons [21], bounded differences [19], simple congruences [14]. It suffices to give specialized algorithms for the assignment operation.

We have realized a prototypical analyzer and, as an application, we have fully developed our technique for the interval domain. The experimental evaluation seems promising, but also shows that there is still space for many improvements. We may choose specific program points where values are collected, such as a loop entry point, in order to better focus the statistical analysis and we may use techniques of code coverage, as in software testing, to improve the quality of execution traces. Moreover, we may partition the set of values we apply PCA to. One idea could be to partition the set of program variables into groups (variables used for array indexes, variables for temporary storage, etc...) which are expected not to be correlated, and perform PCA separately on each group (an idea similar to packing [3]). In addition, we may partition the program code itself (for example around loops), perform a different PCA on each partition, and change the abstract domain appropriately when crossing partitions. In the extreme, we could choose different parameters for each program point, like Sankaranarayanan et al. [25] do for template polyhedra.

The use of linear transformations also suggests to combine PCA with different approaches. We may infer the axes in the new coordinate system from both the semantics and the syntax of the program. The analysis could vastly benefit from the ability to express constraints occurring in the linear expressions of the program, especially in loop guards and array accesses. However, the syntactic approach alone is not recommended, since not all the interesting invariants appear as expressions in the source code. For example, the `cousot78` program does not contain the expressions `i+j`, `j-i` or `2*j-i`: nonetheless, the analysis was able to prove invariants on these constraints (see Figure 8). To overcome this limitation, we may use the probabilistic invariants found by the analysis in [16] instead of using the syntax of the program.

Finally, writing the implementation in R has been useful for rapid prototyping, but porting the code to a faster programming language, possibly within the framework of well known libraries such as `APRON` [18] or `PPL` [2], would make it available to a wider community, while improving performance.

References

1. Anaya-Izquierdo, K., Critchley, F., Vines, K.: Orthogonal simple component analysis. In: Technical Report 08/11, The Open University (2008), http://statistics.open.ac.uk/TechnicalReports/spca_final.pdf (last accessed 2010/03/26)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1-2), 3–21 (2008)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), San Diego, California, USA, June 7-14, pp. 196–207. ACM Press, New York (2003)
4. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Pottosin, I.V., Bjørner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
5. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: Principles Of Programming Languages, POPL 2008 SIGPLAN Not., vol. 43(1), pp. 247–260. ACM, New York (2008)
6. Colóon, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
7. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Paris, France, pp. 106–130. Dunod (1976)
8. Cousot, P., Cousot, R.: Abstract interpretation and applications to logic programs. The Journal of Logic Programming 13(2-3), 103–179 (1992)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 84–97. ACM Press, New York (January 1978)
10. Dor, N., Rodeh, M., Sagiv, M.: Cleanness checking of string manipulations in C programs via integer analysis. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 194–212. Springer, Heidelberg (2001)
11. Feret, J.: Static analysis of digital filters. In: Schmidt [26], pp. 33–48
12. Feret, J.: The arithmetic-geometric progression abstract domain. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 42–58. Springer, Heidelberg (2005)
13. Gopan, D., Reps, T.: Guided static analysis. In: Nielson and Filé [22], pp. 349–365
14. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics 32 (1989)
15. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on *expression abstraction* and *max operator* with application in timing analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 370–384. Springer, Heidelberg (2008)
16. Gulwani, S., Necula, G.C.: Precise interprocedural analysis using random interpretation. In: Principles Of Programming Languages, POPL 2005. SIGPLAN Not., vol. 40(1), pp. 324–337. ACM, New York (2005)

17. Jeannet, B.: Interproc Analyzer for Recursive Programs with Numerical Variables. INRIA. Software and documentation are available at the following, http://pop-art.inrialpes.fr/interproc/interprocweb.cgi (last accessed: 2010-06-11)
18. Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
19. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
20. Minè, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt [26], pp. 3–17
21. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
22. Nielson, H.R., Filé, G. (eds.): SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
23. R Development Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2009)
24. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis using symbolic ranges. In: Nielson and Filé [22], pp. 366–383
25. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
26. Schmidt, D. (ed.): Programming Languages and Systems. ESOP 2004. LNCS, vol. 2986. Springer, Heidelberg (2004)

Concurrent Separation Logic for Pipelined Parallelization

Christian J. Bell, Andrew W. Appel, and David Walker

Princeton University, Computer Science Department,
35 Olden Drive, 08540-5233 Princeton, New Jersey
{cbell,dpw,appel}@cs.princeton.edu
http://www.cs.princeton.edu/

Abstract. Recent innovations in automatic parallelizing compilers are showing impressive speedups on multicore processors using shared memory with asynchronous channels. We have formulated an operational semantics and proved sound a concurrent separation logic to reason about multithreaded programs that communicate asynchronously through channels and share memory. Our logic supports shared channel endpoints (multiple producers and consumers) and introduces *histories* to overcome limitations with local reasoning. We demonstrate how to transform a sequential proof into a parallelized proof that targets the output of the parallelizing optimization DSWP (Decoupled Software Pipelining).

1 Introduction

We have created an operational semantics and a concurrent separation logic (CSL) to reason about the correctness of programs that share memory and use buffered channels to synchronize processes. These channels can be used directly by the programmer or automatically by a parallelizing compiler and are not restricted to point-to-point communication; multiple producers and multiple consumers may asynchronously access a channel. We have proved our CSL sound with respect to the operational semantics. Furthermore, we demonstrate how certain proofs of correctness for a sequential program can be used to generate a related proof (that maintains the original specification) for the parallelized output of an optimization.

CSL [10] is an extension of separation logic (SL), which is an extension of Hoare logic. SL facilitates local reasoning about resources used by a program, so that when analyzing a region of code, we may assume that actions made by the rest of the program cannot interfere. This is encapsulated in the "frame rule" of SL. Similarly, CSL facilitates local reasoning about resources in a concurrent program so that we can analyze just one process while assuming that other processes cannot interfere. CSL has already been used to reason about programs that use critical sections and locks [10][7].

Our logic can be used in proofs about compilers. Leroy and others proved correct compilers for sequential programs; Hobor et al. outlined how such proofs can be extended to concurrent programs [9][7]. We have designed our logic to

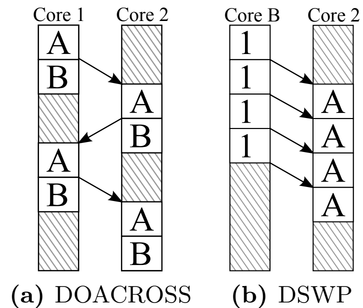
R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 151–166, 2010.
© Springer-Verlag Berlin Heidelberg 2010

be capable of extending these certified compilers to handle programs that use channels. Proofs in our logic can also be used in proof-carrying code frameworks.

1.1 Parallelizing Transformations

A parallelizing compiler attempts to optimize a program by automatically partitioning sequential code into multiple threads. A classic example of this is the DOALL optimization, which parallelizes while-loops that have no loop-carried dependencies by distributing the iterations among multiple threads. While DOALL has had some success, particularly in scientific and numerical computing, it is common for programs to have loop-carried dependencies, thus many programs cannot benefit from DOALL.

Another optimization is DOACROSS, which can handle loop-carried dependencies. This optimization partitions iterations among several threads, but also transmits dependencies between the threads with the hope that there is a significant task within the loop that can overlap with other iterations regardless of the dependency. Figure 1a shows an example trace of a loop where each iteration is composed of tasks A and B; A and B both depend on A, but A and B do not depend on B. Because the dependencies are transmitted bidirectionally between threads, any latency in communication or an iteration stalling will cause the entire computation to stall. Therefore, DOACROSS often does not yield significant performance gains.



(a) DOACROSS **(b)** DSWP

Fig. 1. Trace of DOACROSS vs. DSWP. Arrows depict data flow, control/data dependencies, and communication latency.

Pipelined parallelism identifies code that can be partitioned into tasks that have acyclic dependencies. Tasks that produce dependencies can run ahead of tasks that consume them, and tasks with no dependencies between them run in parallel. Such parallelism is often leveraged at the instruction level in hardware.

Decoupled Software Pipelining (DSWP) is a compiler optimization that leverages pipelined parallelism [15][13], illustrated in Figure 1b. The dependencies are communicated between threads using asynchronous channels, which can be implemented as a shared queue in memory or in hardware [12]. Unlike DOACROSS, communication latencies and stalls will only affect consuming threads, allowing the producing threads to work ahead. DSWP is capable of a significant performance increase: in the SPEC CINT2000 benchmark suite, DSWP yielded a geometric mean speedup of 5.54 with a geometric mean of 17 threads [2].

In Section 2 we will show how to parallelize a wide class of programs. In Section 4 we will show how to transform SL proofs of sequential programs into CSL proofs of DSWP-parallelized programs.

```
while c(p) (
    b(p);
    p:= a(p)
)
```

Fig. 3. Running example

```
while p ≠ nil (
    t:= [p.val];
    [p.val]:= t + 1;
    p:= [p.next]
)
```

Fig. 4. Instance of running example

```
⎛ produce d p;         ║ consume d p';       ⎞
⎜ while c(p) (         ║ while c(p') (       ⎟
⎜     p:= a(p);        ║     b(p');          ⎟
⎜     produce d p      ║     consume d p'    ⎟
⎝ )                    ║ )                   ⎠
```

Fig. 5. Parallelized while-program

2 Parallelizing a Program

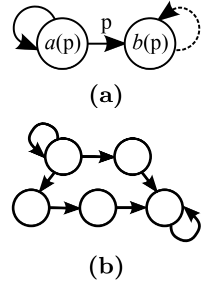


Fig. 2

To illustrate our operational semantics and CSL, we consider the class of programs in Figure 3, where $a(\mathtt{p})$ and $b(\mathtt{p})$ represent blocks of instructions that use at least variable p. We assume operation $a(\mathtt{p})$ does not depend on $b(\mathtt{p})$, $b(\mathtt{p})$ depends only on $a(\mathtt{p})$ exactly through variable p, and $b(\mathtt{p})$ may or may not depend on itself through variables other than p; illustrated in Figure 2a. Figure 4 is an example of such a program. The dependencies in Figure 2a also generalize over more complicated dependencies such as in Figure 2b. In practice, tasks $a(\mathtt{p})$ and $b(\mathtt{p})$ are significant computations.

This program is *not* a candidate for DOALL optimization because of the loop-carried dependencies from $a(\mathtt{p})$ to $a(\mathtt{p})$ and possibly from $b(\mathtt{p})$ to $b(\mathtt{p})$. However, because the dependency between $a(\mathtt{p})$ and $b(\mathtt{p})$ is in one direction, we can apply DSWP to generating the program in Figure 5.

This parallelized program, where we have two processes communicating using instructions `produce` and `consume`, computes the same values as Figure 3. To send the value of expression e through channel d (pushing the value onto the back of the channel's queue), we write `produce` d e. To receive a value (from the front of the channel's queue) into variable x, we write `consume` d x.

3 CSL with Asynchronous Channels

Our logic is an extension of SL with channel endpoints and the heap as resources. We treat channels as a way to transmit both values and resources. While a low-level view of a program may simply see integers being sent through a channel, in our logic those integers may have further meaning. For example, they can be channel identifiers or pointers. When transmitting a pointer, we can bundle the

knowledge that it is a pointer (with permission to dereference it) as a resource, and send the resource along with the pointer value.

To control how the channels are used, we assign resource invariants to each channel to act as a protocol, to which producing and consuming processes must adhere. A resource invariant is a predicate that must always hold on the state of the channel. Resource invariants were introduced to CSL by O'Hearn and have been used to control how critical sections and locks are used [10][7]. Producers use the resource invariant to determine which resources they must give up when sending a value. Consumers use the resource invariant to determine which resources they gain by receiving a value. For example, a resource invariant could state that each value in the channel points to an integer: producers must show that each value sent is such a pointer and give up the resource to dereference it, and consumers use the resource invariant to show that each received value is a valid pointer that can be dereferenced.

3.1 Channel Endpoint Histories

A main property of CSL is that we reason about the correctness of a process independently of all other processes. When a process sends a value to another process through a channel, neither knows what the other has or will do to the value beyond what the resource invariant states. This is enough for memory safety; access to a shared pointer can be transferred between processes to prevent race conditions. However, resource invariants cannot show that values transmitted by a producing process are correctly summed by a consuming process. We also cannot use them to ensure that every pointer sent through a channel is eventually deallocated. Local reasoning in CSL prevents us from reasoning about these behaviors from the perspective of just one process. To overcome this limitation, we delay such reasoning, by recording a *history* of the values transmitted through each endpoint, until the processes synchronize.

Histories in Point-to-Point Communication. Consider two processes: a producer and consumer. The producer sends 1 followed by 2, so its history is $[2, 1]$. The consumer receives two values and stores them into variable x, then y. Because the consumer cannot know what the producer sent, its history is simply $[y, x]$. When the two processes join, we know that $[2, 1]$ was sent and that $[y, x]$ was received, so $[y, x] = [2, 1]$, or $y = 2$ and $x = 1$.

Histories with Multiple Producers and Consumers. We allow a channel endpoint to be split among multiple producers or consumers. Each piece of an endpoint records its own *local* history, and when all the pieces of an endpoint are joined, the history is *global*. Recording histories through a piece of an endpoint is trickier than recording the history in the full endpoint because the order in which the processes send/receive from the channel is nondeterministic. For this reason, we record *sets* of possible histories through each endpoint.

$A ::= \ d_\pi H!$	Share π of the produce endpoint of channel d, with histories H
$\mid d_\pi H?$	Share π of the consume endpoint of channel d, with histories H
$\mid A[\mathsf{p}\colon d \mathrel{+}= v]$	A holds after appending v to d's local produce histories
$\mid A[\mathsf{c}\colon d \mathrel{+}= v]$	A holds after appending v to d's local consume histories
$\mid \mathrm{PHist}\ d\ H$	The local produce histories of channel d are H
$\mid \mathrm{CHist}\ d\ H$	The local consume histories of channel d are H
$\mid e \Downarrow v$	The expression evaluates to value v

$\mid e_1 \mapsto e_2 \quad \mid \quad A_1 * A_2 \quad \mid \quad A_1 \mathrel{-\!\!*} A_2 \quad \mid \quad A_1 \implies A_2 \quad \mid \quad e_1 = e_2 \quad \mid \quad \mathrm{emp} \quad \mid \quad B$

Fig. 6. Predicates

Consider the case of multiple processes producing through the same channel. Each producer records the sequence of the values it sends, but not what other producers send. When two of the processes join, we do not know the order of one history with respect to the other. Thus the combined endpoint records all possible orderings by interleaving the two histories; the actual order in which the values were sent must be within this set. When a new value is subsequently produced through the endpoint, it is appended to each history in the set.

When two sets of histories join, we compute the new set of histories by interleaving every pair of histories from the two sets. We call this operation a *merging* of the histories. Every history in a set of histories is a permutation of all other histories in the set; merging preserves this behavior. For multiple consumers, we record histories in the same way.

Example 1 (Multiple consumers). Assume we have one producing process and two consuming processes, each with an initial set of histories equal to $\{\mathrm{nil}\}$ (none have sent or received any values). The producer sends 1, 2, 3, then 4. Consumer 1 receives two values, w then x; consumer 2 receives one value, y; then they join and one more value, z, is consumed. The resulting sets of histories are:

Producer	Consumer 1	Consumer 2	After C1 & C2 join, then consume z
$\{[4,3,2,1]\}$	$\{[x,w]\}$	$\{[y]\}$	$z{::}(\{[x,w]\} \mathbin{/\!\!\!\wedge\!\!\!\backslash} \{[y]\}) =$ $z{::}\{[x,w,y],[x,y,w],[y,x,w]\} =$ $\{[z,x,w,y],[z,x,y,w],[z,y,x,w]\}$

The merged ($\mathbin{/\!\!\!\wedge\!\!\!\backslash}$) consume histories cannot exactly reconstruct the order in which the two consumers received values with respect to each other. They can show, however, that $z = 4$ and $z > x > w$.

3.2 Predicate Logic

Figure 6 lists some predicates used in our logic. Metavariable A is a predicate that ranges over formulae, e is an expression, H is a set of histories (each a list of values), and d is a channel name. π is a fractional share of a resource that ranges over $[0, 1]$, where $\pi > 0$ grants permission for [shared] use of the resource

and $\pi = 0$ does not. A value is either an integer or list of values. The predicates in the last line are conventional: separating conjunction, separating implication, implication, expression equality, no share of any resources, and a pure logical formula. The logic has universal and existential quantification ranging over values. We use the forcing relation $r; s \models A$ to state that predicate A holds under environment s and exactly resources r (resources are defined in Section 5.1). Predicate A_1 entails A_2 if $\forall r, s.\ r; s \models A_1 \implies r; s \models A_2$, written $A_1 \vdash A_2$.

Predicates PHist/CHist hold for any resource if the produce/consume history is exactly H. They are used as side conditions for some of the Hoare rules.

Channel endpoints interact with the separating conjunction as follows:

$$\begin{aligned} d_{\pi_1} H_1! * d_{\pi_2} H_2! \quad &\iff \quad d_{\pi_1+\pi_2}(H_1 \mathbin{/\!\!\backslash\!\backslash} H_2)! \\ d_{\pi_1} H_1? * d_{\pi_2} H_2? \quad &\iff \quad d_{\pi_1+\pi_2}(H_1 \mathbin{/\!\!\backslash\!\backslash} H_2)? \end{aligned}$$

Example 2. Assume that Example 1 uses channel d and the two consumers have permissions π_1 and π_2. Just before the consumers join, they have resources $d_{\pi_1}\{[x, w]\}?$ and $d_{\pi_2}\{[y]\}?$ respectively. After joining, their resources are:

$$\begin{aligned} d_{\pi_1}\{[x, w]\}? * d_{\pi_2}\{[y]\}? &\vdash d_{\pi_1+\pi_2}(\{[x, w]\} \mathbin{/\!\!\backslash\!\backslash} \{[y]\})? \\ &\vdash d_{\pi_1+\pi_2}\{[x, w, y], [x, y, w], [y, x, w]\}? \end{aligned}$$

And after consuming into variable z, their resources are:

$$\begin{aligned} &d_{\pi_1+\pi_2}(z{::}\{[x, w, y], [x, y, w], [y, x, w]\})? \\ \vdash\ &d_{\pi_1+\pi_2}\{[z, x, w, y], [z, x, y, w], [z, y, x, w]\}? \end{aligned}$$

3.3 Resource Invariants

We use resource invariants to verify that values and resources are transmitted through each channel according to protocol. The state of a channel is composed of the resources r stored in the channel, the list of values l_q queued in the channel, and the list of previously consumed values l_c (not a *set* of histories). A resource invariant R holds on the state (r, l_q, l_c) of a channel only if $r; . \models R(l_q, l_c)$. Concatenating the queue l_q and consume values l_c of a channel together, written $l_q@l_c$, yields the list of produced values for the channel. Channel resource invariants must satisfy the predicate $\vdash R$ `R-okay`, specified as follows:

$$\frac{\begin{array}{ll} \text{emp} & \vdash R(\text{nil}, \text{nil}) \\ \forall l_q, l_c.\ R(l_q, l_c) & \text{is } \textit{closed} \text{ and } \textit{precise} \\ \forall l_c.\ R(\text{nil}, l_c) & \vdash \text{emp} \end{array}}{\vdash R\ \texttt{R-okay}}\ \text{Resource-Okay}$$

The first premise states that the resource invariant must be satisfied and own no resources for a channel that has not yet been used. The second ensures that resources can be transferred between process environments and that the resource invariant is sufficient to determine exactly what resources are transferred to and

$$\begin{array}{l} \iota ::= \iota_1; \iota_2 \;\mid\; x \texttt{:=}\; e \;\mid\; x \texttt{:= [}e\texttt{]} \;\mid\; \texttt{[}x\texttt{]:=}\; e \;\mid\; \texttt{while}\; e\; \iota \;\mid\; \texttt{assert}\; A \\ \quad\mid\; [\Gamma_1; A_1]\; \iota_1 \parallel [\Gamma_2; A_2]\; \iota_2 \;\mid\; \texttt{produce}\; d\; e \;\mid\; \texttt{consume}\; d\; x \;\mid\; \texttt{skip} \end{array}$$

Instruction sequencing, local variable assignment, fetch from heap, store into heap, repeat ι until e is false, assert that predicate A holds, run two blocks of instructions in parallel, produce a value, consume a value into a local variable, no-op.

Fig. 7. Instructions

from the channel. Finally, the third premise attaches resources only to values in the queue and not the consume history to ensure that all resources can eventually be extracted from the channel by consuming values.

Example 3. A resource invariant that specifies the first value produced/consumed is 0 and that the values transmitted are strictly increasing:

$$\begin{aligned} R \triangleq \lambda l_q.\lambda l_c.\ \text{match } l_q, l_c \text{ with} \\ &\mid \text{nil}, \text{nil} \Rightarrow \text{emp} \\ &\mid \text{nil}, v\text{::nil} \Rightarrow v = 0 \wedge \text{emp} \\ &\mid \text{nil}, v_1\text{::}v_2\text{::}l_c \Rightarrow v_1 > v_2 \wedge R(\text{nil}, v_2\text{::}l_c) \\ &\mid l_q @ v\text{::nil}, l_c \Rightarrow R(l_q, v\text{::}l_c) \end{aligned}$$

Example 4. A resource invariant for the parallelization of Figure 4 in Figure 5. To pass permission to dereference `p.val` through the channel:

$$\begin{aligned} R \triangleq \lambda l_q.\lambda l_c.\ \text{match } l_q, l_c \text{ with} \\ &\mid \text{nil}, _ \Rightarrow \text{emp} \\ &\mid v\text{::}l_q', _ \Rightarrow R(l_q', l_c) * v.\texttt{val} \mapsto - \end{aligned}$$

3.4 Instructions

In Figure 7, x is a program variable, ι is an instruction, A is a predicate, Γ is a set of free variables, and d is the name of a channel. Instruction $[\Gamma_1; A_1]\ \iota_1 \parallel [\Gamma_2; A_2]\ \iota_2$ uses Γ and A to specify how variables and resources are split between processes ι_1 and ι_2. We use `assert` to prove the partial correctness of programs.

3.5 Hoare Logic

A Hoare triple describes the precondition and postcondition of executing a command. If the precondition is met and the command terminates, then the postcondition establishes the new state of the program. We give the Hoare triple rules for our logic in Figure 8. $FV(X)$ denotes the set of free variables in X, where X ranges over instructions and predicates. The Hoare triple $\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \iota\ \{B\}$ is composed of an instruction ι, precondition A, postcondition B, environment

$$l \Subset_\pi H \triangleq \begin{cases} l \in H & \text{if } \pi = 1 \\ \exists H'.\ l \in H \wedge\!\!\!\wedge H' & \text{if } \pi \neq 1 \end{cases}$$

$$\frac{\begin{array}{l} B \vdash \text{PHist } d\ \{\text{nil}\} \qquad A \vdash \text{PHist } d\ e{::}H \\ \forall l_q, l_c.\ \ l_q @ l_c \Subset_\pi H \implies B * \bar{R}[d](l_q, l_c) \vdash \bar{R}[d](e{::}l_q, l_c) \end{array}}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{d_\pi H! * (d_\pi e{::}H! \mathrel{-\!\!*} B * A)\}\ \texttt{produce}\ d\ e\ \{A\}}\ \text{H-Produce}$$

$$\frac{\begin{array}{l} x \in \Gamma \quad \forall v.\ B(v) \vdash \text{CHist } d\ \{\text{nil}\} \quad A \vdash \text{CHist } d\ x{::}H \\ \forall l_q, v, l_c.\ \ l_c \Subset_\pi H \implies \\ \qquad \bar{R}[d](l_q @ v{::}\text{nil}, l_c) \vdash B(v) * \bar{R}[d](l_q, v{::}l_c) \end{array}}{\begin{array}{ll} \bar{R}; \Gamma \vdash_{\mathrm{i}} & \{d_\pi H? * (\forall v.\ (d_\pi H? * B(v))[\mathsf{c}\colon d \mathrel{+}= v] \mathrel{-\!\!*} A[v/x])\} \\ & \texttt{consume}\ d\ x \\ & \{A\} \end{array}}\ \text{H-Consume}$$

$$\frac{\begin{array}{lll} \bar{R}; \Gamma_1 \vdash_{\mathrm{i}} \{A_1\}\ \iota_1\ \{B_1\} & FV(A_1) \subseteq \Gamma_1 & \Gamma_1 \# \Gamma_2 \\ \bar{R}; \Gamma_2 \vdash_{\mathrm{i}} \{A_2\}\ \iota_2\ \{B_2\} & FV(A_2) \subseteq \Gamma_2 & \Gamma_1 \cup \Gamma_2 \subseteq \Gamma \end{array}}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A_1 * A_2\}\ [\Gamma_1; A_1]\ \iota_1 \parallel [\Gamma_2; A_2]\ \iota_2\ \{B_1 * B_2\}}\ \text{H-Parallel}$$

$$\frac{x \in \Gamma}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{\exists v.\ e \mapsto v * (e \mapsto v \mathrel{-\!\!*} A[v/x])\}\ x \texttt{:= [}e\texttt{]}\ \{A\}}\ \text{H-Fetch}$$

$$\frac{}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{e_1 \mapsto - * (e_1 \mapsto e_2 \mathrel{-\!\!*} A)\}\ \texttt{[}e_1\texttt{]:=}\ e_2\ \{A\}}\ \text{H-Store}$$

$$\frac{}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \texttt{assert}\ A\ \{A\}}\ \text{H-Assert} \qquad \frac{x \in \Gamma}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A[e/x]\}\ x \texttt{:=}\ e\ \{A\}}\ \text{H-Assign}$$

$$\frac{\begin{array}{l} \bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \iota\ \{B\} \qquad FV(\iota) \cap FV(C) = \emptyset \\ \forall d \in FV(\iota).\ C \vdash \text{PHist } d\ \{\text{nil}\} \wedge \text{CHist } d\ \{\text{nil}\} \end{array}}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A * C\}\ \iota\ \{B * C\}}\ \text{H-Frame}$$

$$\frac{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A \wedge e\}\ i\ \{A\}}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \texttt{while}\ e\ i\ \{A \wedge \neg e\}}\ \text{H-While}$$

$$\frac{\begin{array}{l} A \vdash A' \qquad B' \vdash B \\ \bar{R}; \Gamma \vdash_{\mathrm{i}} \{A'\}\ \iota\ \{B'\} \end{array}}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \iota\ \{B\}}\ \text{H-Consequence} \qquad \frac{\begin{array}{l} \bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \iota_1\ \{B\} \\ \bar{R}; \Gamma \vdash_{\mathrm{i}} \{B\}\ \iota_2\ \{C\} \end{array}}{\bar{R}; \Gamma \vdash_{\mathrm{i}} \{A\}\ \iota_1 \texttt{;} \iota_2\ \{B\}}\ \text{H-Seq}$$

Fig. 8. Inference rules of the Hoare logic

domain Γ, and a channel-indexed list of resource invariants $\bar{R}$. We write $\Gamma_1 \# \Gamma_2$ if the two domains are disjoint. For any element X in our formal system, we write $\bar{X}$ to denote a list of such elements and $\bar{X}[i]$ to access the ith element.

H-Produce. Consider a program that produces pointers to nodes in a linked list with the intention that the consumer only accesses the val member of each node, such as in Figure 5. To execute `produce` d `p` and express that $\mathsf{p.val} \mapsto -$ is transferred to the channel (to eventually be transferred to a consumer), that the produce histories are appended with p, and that the resource $\mathsf{p.next} \mapsto -$ is retained, we could use the Hoare triple and resource invariant:

$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{d_1 H! * \mathsf{p.val} \mapsto - * \mathsf{p.next} \mapsto -\} \ \mathtt{produce}\ d\ \mathtt{p}\ \{d_1 \mathsf{p}{::}H! * \mathsf{p.next} \mapsto -\}$$

$$\begin{aligned} \bar{R}[d] \triangleq \lambda l_q . \lambda l_c .\ & \text{match } l_q, l_c \text{ with} \\ & \mid \text{nil}, _ \Rightarrow \text{emp} \\ & \mid v{::}l'_q, _ \Rightarrow R(l'_q, l_c) * v.\mathtt{val} \mapsto - \end{aligned}$$

To prove this triple, we apply rules H-Produce and H-Consequence and prove these side conditions:

$$\forall l_q, l_c.\ l_q @ l_c \Subset_1 H \implies \big(\mathsf{p.val} \mapsto - * R(l_q, l_c) \vdash R(\mathsf{p}{::}l_q, l_c)\big) \quad (1)$$

$$\mathsf{p.val} \mapsto - \vdash \text{PHist } d\ \{\text{nil}\} \quad (2)$$

$$d_1 \mathsf{p}{::}H! * \mathsf{p.next} \mapsto - \vdash \text{PHist } d\ \mathsf{p}{::}H \quad (3)$$

$$\begin{pmatrix} d_1 H! * \mathsf{p.val} \mapsto - \\ * \ \mathsf{p.next} \mapsto - \end{pmatrix} \vdash \begin{pmatrix} d_1 H! * \big(d_1 \mathsf{p}{::}H! \mathrel{-\!\!*} (\mathsf{p.val} \mapsto - \\ \qquad * \ d_1 \mathsf{p}{::}H! * \mathsf{p.next} \mapsto -)\big) \end{pmatrix} \quad (4)$$

These obligations are easy to prove for this example. (The antecedent in obligation 1 can be ignored because it is not needed to prove the consequent).

Rule H-Produce requires some permission ($d_\pi H!$) to access the produce endpoint and implies that the post state will have histories H appended with the value sent. The first judgement in H-Produce, $B \vdash$ PHist d $\{\text{nil}\}$, prevents the resources B that are transferred to the channel from specifying histories for the same channel. Judgement $A \vdash$ PHist d $e{::}H$ prevents the postcondition from specifying histories in addition to $e{::}H$. These side conditions involving PHist and CHist are necessary to prove soundness using our current model of histories. They do not prevent channels from sending shares of themselves (with histories $\{\text{nil}\}$) or shares and histories of other channels.

The third judgement requires that, for any state of the channel (the queue l_q and consumed values l_c) such that the resource invariant holds, the invariant remains satisfied after pushing e onto the back of the queue and adding resources B to the channel. Its antecedent, $l_q @ l_c \Subset_\pi H$, restricts the set of channel states to those that support the local histories we have observed.

When $\pi = 1$, $l_q @ l_c \Subset_\pi H$ simplifies to $l_q @ l_c \in H$ (H contains all possible orderings of produced values), and when $\pi < 1$, it implies that every value in H is present in the list of produced values $l_q @ l_c$. This constraint, although weak, still has uses. For example, it allows producing a 0 to a channel with the resource

invariant "all values must be 1 until a 0 is produced, at which point only 0's can be produced" if all we know is that a 0 appears in the local produce histories.

H-Consume. Consider the consumer process from the previous example program, which consumes values that are pointers to nodes in a linked list, only to dereference the val member of each node. To execute `consume` d `p'`, express that $\texttt{p'}.\mathsf{val} \mapsto -$ is received from the channel, and that the consume histories are appended with `p'`, we could use the Hoare triple:

$$\bar{R}; \Gamma, x \vdash_{\mathrm{i}} \{d_1 H?\}\ \texttt{consume}\ d\ \texttt{p'}\ \{d_1 \texttt{p'}{::}H? * \texttt{p'}.\mathsf{val} \mapsto -\}$$

To prove this triple, we apply rules H-Consume and H-Consequence and prove:

$$\forall l_q, v, l_c.\ l_c \Subset_1 H \implies \big(R(l_q @ v{::}\mathrm{nil}, l_c) \vdash v.\mathsf{val} \mapsto - * R(l_q, v{::}l_c)\big) \tag{5}$$

$$v.\mathsf{val} \mapsto - \vdash \mathrm{CHist}\ d\ \{\mathrm{nil}\} \tag{6}$$

$$d_1 \texttt{p'}{::}H? * \texttt{p'}.\mathsf{val} \mapsto - \vdash \mathrm{CHist}\ d\ \texttt{p'}{::}H \tag{7}$$

$$d_1 H? \vdash \begin{pmatrix} d_1 H? * \big(\forall v.\ (d_1 H? * v.\mathsf{val} \mapsto -)[\mathsf{c}\colon d{+}{=}\ v] \\ \quad -\!\!* (d_1 \texttt{p'}{::}H? * \texttt{p'}.\mathsf{val} \mapsto -)[v/\texttt{p'}]\big) \end{pmatrix} \tag{8}$$

These particular obligations are also easy to prove. Obligation 8, although convoluted, can be deduced easily:

$$d_1 H? \vdash \begin{pmatrix} d_1 H? * \big(\forall v.\ (d_1 H? * v.\mathsf{val} \mapsto -)[\mathsf{c}\colon d{+}{=}\ v] \\ \quad -\!\!* (d_1 \texttt{p'}{::}H? * \texttt{p'}.\mathsf{val} \mapsto -)[v/\texttt{p'}]\big) \end{pmatrix}$$

$$emp \vdash \begin{pmatrix} \forall v.\ (d_1 H? * v.\mathsf{val} \mapsto -)[\mathsf{c}\colon d{+}{=}\ v] \\ -\!\!* (d_1 \texttt{p'}{::}H? * \texttt{p'}.\mathsf{val} \mapsto -)[v/\texttt{p'}]) \end{pmatrix}$$

$$(d_1 H? * v.\mathsf{val} \mapsto -)[\mathsf{c}\colon d{+}{=}\ v] \vdash (d_1 \texttt{p'}{::}H? * \texttt{p'}.\mathsf{val} \mapsto -)[v/\texttt{p'}]$$

$$d_1 v{::}H? * v.\mathsf{val} \mapsto - \vdash d_1 v{::}H? * v.\mathsf{val} \mapsto -$$

Rule H-Consume requires permission ($d_\pi H?$) to access the consume endpoint. The fourth judgement in H-Consume requires that, for all states of the channel (the queue $l_q @ v :: \mathrm{nil}$ and consumed values l_c) such that its resource invariant holds, the invariant remains satisfied after popping v from the front of the queue, recording it in the list of consumed values, and removing the resources $B(v)$ tied to the value. (B is a function from values to predicates). The antecedant, $l_c \Subset_\pi H$, restricts the consumed values to those that are supported by the local consume histories. Resources $B(v)$ are transferred to the consuming process. Finally, the local consume histories are appended with the consumed value.

4 Parallelized Program with Proof

Consider proofs of correctness for programs such as Figure 3 (and particularly Figure 4) that follow the schema in Figure 9. We give an example instance of such

$\bar{R}; \Gamma \vdash_\iota$
$\{C(\text{nil}, \mathtt{p}) * F(\mathtt{p}) * D(\text{nil})\}$
$\{\exists h.\ C(h, \mathtt{p}) * F(\mathtt{p}) * D(h)\}$
`while` c(p) (
$\{C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\}$
b(p) ;
$\{C(h, \mathtt{p}) * D(\mathtt{p}{::}h) \wedge c(\mathtt{p}) \wedge \mathtt{p}\Downarrow v\}$
p:= a(p)
$\{C(v{::}h, \mathtt{p}) * F(\mathtt{p}) * D(v{::}h)\}$
$\{\exists h.\ C(h, \mathtt{p}) * F(\mathtt{p}) * D(h)\}$
)
$\{\exists h.\ C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge \neg c(\mathtt{p})\}$

Fig. 9. While-program with proof

$$C(l, v) \triangleq \mathtt{plist}\ l^{\mathrm{R}}\ v * \exists l'.\ j = l^{\mathrm{R}}@l' \wedge (v \neq \text{nil} \iff v = \mathtt{hd}\ l') \wedge (F(v) \mathrel{-\!\!*} \mathtt{list}\ l'\ \text{nil})$$
$$F(v) \triangleq \text{if } v \neq \text{nil then } v.\mathsf{val} \mapsto - \text{ else emp}$$
$$D(l) \triangleq \mathtt{vlist}\ l$$

$\mathtt{plist}\ l\ v \triangleq \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{emp}$
$\mid x{::}\text{nil} \Rightarrow x.\mathsf{next} \mapsto v \wedge x \neq \text{nil}$
$\mid w{::}x{::}l \Rightarrow w.\mathsf{next} \mapsto x * \mathtt{plist}\ x{::}l\ v \wedge w \neq \text{nil}$
$\mathtt{vlist}\ l \triangleq \text{match } l \text{ with } \mid \text{nil} \Rightarrow \text{emp}$
$\mid x{::}l' \Rightarrow x.\mathsf{val} \mapsto - * \mathtt{vlist}\ l' \wedge x \neq \text{nil}$
$\mathtt{list}\ l\ t \triangleq \mathtt{vlist}\ l * \mathtt{plist}\ l\ t$

Fig. 10

a proof in Figure 10 for the program in Figure 4. The example proof ensures that the shape and order of the linked list is preserved, and holds for the properties:

$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{C(h, \mathtt{p}) * D(h) \wedge c(\mathtt{p}) \wedge \mathtt{p}\Downarrow v\}\ \mathtt{p}\mathtt{:=}\ a(\mathtt{p})\ \{F(\mathtt{p}) * D(h) * C(v{::}h, \mathtt{p}) \wedge c(v)\} \tag{9}$$

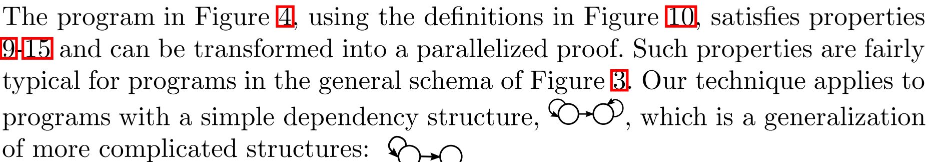
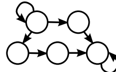
$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{C(h, \mathtt{p}) * F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\}\ b(\mathtt{p})\ \{C(h, \mathtt{p}) * D(\mathtt{p}{::}h) \wedge c(\mathtt{p})\} \tag{10}$$

Furthermore, if the following properties hold, then we can construct a proof for programs characterized by Figure 3 and optimized by DSWP:

$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{C(h, \mathtt{p}) \wedge c(\mathtt{p}) \wedge \mathtt{p}\Downarrow v\}\ \mathtt{p}\mathtt{:=}\ a(\mathtt{p})\ \{F(\mathtt{p}) * C(v{::}h, \mathtt{p}) \wedge c(v6)\} \tag{11}$$
$$\bar{R}; \Gamma \vdash_{\mathrm{i}} \{F(\mathtt{p}) * D(h) \wedge c(\mathtt{p})\}\ b(\mathtt{p})\ \{D(\mathtt{p}{::}h) \wedge c(\mathtt{p})\} \tag{12}$$
$$\mathrm{FV}(b(\mathtt{p})) \cap \mathrm{MV}(\mathtt{p}\mathtt{:=}\ a(\mathtt{p})) = \mathtt{p} \tag{13}$$
$$\mathrm{FV}(c(\mathtt{p})) \cap \mathrm{MV}(\mathtt{p}\mathtt{:=}\ a(\mathtt{p})) = \mathtt{p} \tag{14}$$
$$\mathrm{FV}(c(\mathtt{p})) \cap \mathrm{MV}(b(\mathtt{p})) = \emptyset \tag{15}$$

Theorem 1. *Given any program instance of Figure 3 with a (sequential) Separation Logic proof given by (9) and (10), and satisfying (11)-(15), there is a Concurrent Separation Logic proof of the correctness of the parallelized program instance in Figure 5.*

Proof. See technical report[1].

The program in Figure 4, using the definitions in Figure 10, satisfies properties 9-15 and can be transformed into a parallelized proof. Such properties are fairly typical for programs in the general schema of Figure 3. Our technique applies to programs with a simple dependency structure, , which is a generalization of more complicated structures: 

$$\pi \in \mathsf{Share} \triangleq [0,1] \qquad u \in \mathsf{Location}$$
$$d \in \mathsf{ChannelName} \qquad H \in \mathsf{HistorySet}$$

$$\begin{aligned}\mathsf{Endpoint} &\triangleq \{\, (\pi, H) : \mathsf{Share} \times \mathsf{HistorySet} \mid \pi = 0 \implies H = \{\text{nil}\}\} \\ r &: \{\, \mathsf{m} : \mathsf{Location} \rightarrow \mathsf{option\ value}, \\ &\quad\ \mathsf{p} : \mathsf{ChannelName} \rightarrow \mathsf{option\ Endpoint}, \\ &\quad\ \mathsf{c} : \mathsf{ChannelName} \rightarrow \mathsf{option\ Endpoint}\}\end{aligned}$$

$$\frac{\forall x.\ f_1(x) \oplus f_2(x) \ = \ f_3(x)}{f_1 \oplus_f f_2 \ = \ f_3}\ \text{Join-Function}$$

$$\frac{\begin{array}{c}\pi_1 \oplus_s \pi_2 \ = \ \pi_3 \\ H_1 \mathbin{\wedge\!\!\!\wedge\!\!\!\wedge} H_2 \ = \ H_3\end{array}}{(\pi_1, H_1) \oplus_e (\pi_2, H_2) \ = \ (\pi_3, H_3)}\ \text{Join-Endpoint} \qquad \frac{\begin{array}{c}r_1.\mathsf{m} \oplus_f r_2.\mathsf{m} \ = \ r_3.\mathsf{m} \\ r_1.\mathsf{p} \oplus_f r_2.\mathsf{p} \ = \ r_3.\mathsf{p} \\ r_1.\mathsf{c} \oplus_f r_2.\mathsf{c} \ = \ r_3.\mathsf{c}\end{array}}{r_1 \oplus_w r_2 \ = \ r_3}\ \text{Join-World}$$

Fig. 11. Channel Resources

5 Model and Operational Semantics

5.1 The Separation Algebra of Resources

A standard technique [3] for constructing a SL is to first construct a Separation Algebra (SA) on the resources. Our CSL is based on the SA of *worlds*, which are composed of three kinds of resources: the heap, produce endpoints, and consume endpoints. Although the heap is not necessary to demonstrate a CSL for channels, we include it to prove the correctness of parallelized pointer-programs, where channels are used to synchronize access to shared memory and prevent race conditions. We have formalized[1] our SA following Dockins et al. [4].

A SA is a tuple, $\langle E, \oplus \rangle$, where E is some type and $\oplus$ is a "join" relation that satisfies functionality, associativity, commutativity, cancellation, self-join ($a \oplus a = b \implies a = b$), and has a unit for each element ($\forall a. \exists e.\ e \oplus a = a$).

We first define worlds, r, as a record of the heap (m), produce endpoints (p), and consume endpoints (c). These worlds and the join relation $\oplus_w$ (rule Join-World in Figure 11) form a SA. Dockins et al. have shown how to construct a SA for the heap and $\oplus_f$; we have constructed a SA for channel endpoints using the same approach, using the fact that $\langle \mathsf{Endpoint}, \oplus_e \rangle$ is a SA.

A Share is the set of rationals over $[0, 1]$. The relation $\oplus_s$ is the addition operator, and share $\pi = 0$ is the unit for $\oplus_s$. A channel endpoint equal to Some (π, H) implies $\pi > 0$; None implies a share of 0. In our notation, the produce history of channel d, in world r, is $r.\mathsf{p}(d).\mathsf{H}$; its share is $r.\mathsf{p}(d).\pi$; the consume history is $r.\mathsf{c}(d).\mathsf{H}$; and the consume share is $r.\mathsf{c}(d).\pi$. (This notation implies that $r.\mathsf{p}(d) \neq$ None and $r.\mathsf{c}(d) \neq$ None). We write $r.\mathsf{m}(u) =$ Some v if the heap in world r contains value v at address u.

[1] A Coq formalization of our SA is available at
`http://www.cs.princeton.edu/~cbell/cslchannels/`

$$
\begin{array}{rcl}
r; s \models d_\pi H! & \text{iff} & r.\mathsf{p}(d) = \text{Some } (\pi, [\![H]\!]_s) \land \forall d' \neq d.\ r.\mathsf{p}(d') = \text{None} \\
& & \quad \land \forall d.\ r.\mathsf{c}(d) = \text{None} \land \forall l.\ r.\mathsf{m}(l) = \text{None} \\
r; s \models d_\pi H? & \text{iff} & r.\mathsf{c}(d) = \text{Some } (\pi, [\![H]\!]_s) \land \forall d' \neq d.\ r.\mathsf{c}(d') = \text{None} \\
& & \quad \land \forall d.\ r.\mathsf{p}(d) = \text{None} \land \forall l.\ r.\mathsf{m}(l) = \text{None} \\
r; s \models A[\mathsf{p}\text{: } d \mathrel{+}= v] & \text{iff} & \exists r'.\ r = r'[\mathsf{p}\text{: } d \mathrel{+}= v] \land r'; s \models A \\
r; s \models A[\mathsf{c}\text{: } d \mathrel{+}= v] & \text{iff} & \exists r'.\ r = r'[\mathsf{c}\text{: } d \mathrel{+}= v] \land r'; s \models A \\
r; s \models \text{PHist } d\ H & \text{iff} & r.\mathsf{p}(d).\mathsf{H} = [\![H]\!]_s \\
r; s \models \text{CHist } d\ H & \text{iff} & r.\mathsf{c}(d).\mathsf{H} = [\![H]\!]_s \\
r; s \models e_1 \mapsto e_2 & \text{iff} & r.\mathsf{m}([\![e_1]\!]_s) = \text{Some } [\![e_2]\!]_s \land \forall l \neq [\![e_1]\!]_s.\ r.\mathsf{m}(l) = \text{None} \\
& & \quad \land \forall d.\ r.\mathsf{p}(d) = \text{None} \land r.\mathsf{c}(d) = \text{None} \\
r; s \models A_1 * A_2 & \text{iff} & \exists r_1, r_2.\ r_1 \oplus r_2 = r \land r_1; s \models A_1 \land r_2; s \models A_2 \\
r; s \models A_1 \mathrel{-\!\!*} A_2 & \text{iff} & \forall r_1, r_2.\ r_1; s \models A_1 \implies r \oplus r_1 = r_2 \implies r_2; s \models A_2 \\
r; s \models A_1 \implies A_2 & \text{iff} & r; s \models A_1 \implies r; s \models A_2 \\
r; s \models e_1 = e_2 & \text{iff} & [\![e_1]\!]_s = [\![e_2]\!]_s \\
r; s \models \text{emp} & \text{iff} & \forall l.\ r.\mathsf{m}(l) = \text{None} \land \forall d.\ r.\mathsf{p}(d) = \text{None} \land r.\mathsf{c}(d) = \text{None}
\end{array}
$$

Fig. 12. Predicate Formulae

A HistorySet is a nonempty set of histories. The merge function ($\curlywedge\!\!\!\!\curlywedge$) satisfies all the properties of a SA except self-join. Constructing a SA from the tuple of two SAs is straightforward, but without the property of self-join for $\curlywedge\!\!\!\!\curlywedge$, we must add the condition that for any Endpoint (π, H), if the share empty $(\pi = 0)$, then the set of histories is $\{\text{nil}\}$.

5.2 Predicate Formulae

In Figure 12, we write $[\![e]\!]_s$ to evaluate e within environment s, $r[\mathsf{p}\text{: } d \mathrel{+}= v]$ to append value v to the local produce histories of channel d in r, and $r[\mathsf{c}\text{: } d \mathrel{+}= v]$ to similarly update the local consume histories. An environment, s, is a finite partial map from variables to values.

5.3 Operational Semantics

The machine state is the tuple $S = (\bar{c}, \bar{P})$, where $\bar{c}$ is a channel-indexed list of tuples which contain information about each channel's state. Specifically, $\bar{c}[d] = (R, r, l_q, l_c)$, where R is the resource invariant, r are the resources contained within the queue, l_q is the queue, and l_c is the list of consumed values. $\bar{P}$ is a list of processes, where for any process k, $\bar{P}[k] = (r, s, z)$, r are the process' resources, s is its environment, and z is the current instruction. Stepping process k from state S to state S' is written as $S \rightarrow_k S'$.

Our operational semantics use permissions to guarantee that programs are well-synchronized, so that any process will get stuck if it attempts to access a resource without permission. Crucially, two processes may not have permission to mutate the same memory at the same time. Well-synchronized programs are race free, and it is generally understood that such programs have equivalent executions in both strong and weak memory models. Thus, while our operational semantics define a strong memory model, using a weak memory model would not change its behavior.

6 Soundness

We prove[2] soundness for our logic using progress and preservation. In our proofs, we consider only well-formed machine states S such that $\vdash S$ `well-formed`. A well-formed machine state requires each channel state to satisfy its resource invariant; that the resource invariant R for each channel is $\vdash R$ `R-okay`; that each process is well-formed; and that no two processes share any of the same environment variables. A process is well-formed only if its instructions have a Hoare triple derivation and if the precondition of this triple is exactly satisfied by the process' current resources and environment.

For any machine-states S and S',

Theorem 2 (Preservation). *For all processes k, if $\vdash S$ well-formed and $S \rightarrow_k S'$, then $\vdash S'$ well-formed.*

Theorem 3 (Progress). *If $\vdash S$ well-formed, then for all processes k, there either exists a state S' such that $S \rightarrow_k S'$, or process k in S is halted.*

7 Related Work

Ottoni informally proved that when parallelizing a program, if the dependencies (the Program Dependency Graph) are preserved, then the generated program is observationally equivalent [11]. Yet this is not enough for a certified compiler because no implementation of the algorithm is proven.

Our CSL with channels was partially modeled after Concurrent C Minor [7]; specifically the style of resource invariants. There are close similarities between our while-programs with channels and pi-calculus, enough so that our CSL is similar to the logic proposed by Hoare and O'Hearn [6]. Turon et al. have extended that work with multiple producers and consumers [14] independently from us. Neither of these works have shared memory or an equivalent to histories, so are not applicable to reasoning about the correctness of DSWP.

Hurlin shows how to parallelize a sequential program's proof, using rewrite rules on its derivation tree, for the DOALL optimization [8]. We demonstrate how to prove a DSWP-parallelized program without using the proof's derivation tree, but we assume a proof structure that may not characterize all proofs.

---

[2] Our proofs can be found at `http://www.cs.princeton.edu/~cbell/cslchannels/`

8 Future Work and Conclusion

We have developed an operational semantics and CSL for asynchronous channels and proved the logic sound. This logic features histories that are used to overcome limitations in local reasoning and to prove the correctness of parallelized programs in the presence of asynchronous communication. We demonstrated how an existing proof of correctness (of a particular structure) for a sequential program can be used to generate a related proof for the parallelized output of DSWP. By maintaining the preconditions and postconditions, we can prove partial correctness of the optimization.

Our CSL and operational semantics are thus potential targets for an automatic method of generating parallelized proofs for parallelizing optimizations. Using such a method, we could prove that the optimization preserves all *specified* behaviors of a program.

First class channels are unnecessary for our proofs about DSWP. Some separation logics have first class objects [5][7], others do not permit passing objects this way [10]. We would need allocation, deallocation, and a more complicated definition of histories in order to support first class channels. The first two are not possible in first-order logic, but we believe it is straightforward to add following the approach used by Gotsman et al. and Hobor et al. The more complicated history model requires histories to have shared pasts that, upon merging, would retain their original ordering. For example, breaking the history $\{[123]\}$ into histories $\{[12]\}$ and $\{[3]\}$ would result in $\{[123]\}$ upon re-merging rather than $\{[123], [132], [312]\}$. (With such histories, side conditions involving PHist and CHist could be dropped from the Hoare rules).

We are now investigating methods of manipulating an existing arbitrary proof of a program using facts acquired from shape analysis, with the goal of eventually proving the correctness of DSWP and other parallelizing optimizations.

Acknowledgements. This research is funded in part by NSF awards CNS-0627650 and IIS-0612147. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Bell, C.J., Appel, A.W., Walker, D.: Concurrent Separation Logic for Pipelined Parallelization (2010), `http://www.cs.princeton.edu/cbell/cslchannels/cslchannels_techreport.pdf`
2. Bridges, M.J., Vachharajani, N., Zhang, Y., Jablin, T., August, D.I.: Revisiting the Sequential Programming Model for Multi-Core. In: Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 69–81 (December 2007)
3. Calcagno, C., O'Hearn, P., Yang, H.: Local actions and abstract separation logic. In: Proceeding of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 353–367 (2008)

4. Dockins, R., Hobor, A., Appel, A.W.: A Fresh Look at separation algebras and Share Accounting. In: 7th Asian Symposium on Programming Languages and Systems. Springer ENTCS (December 2009)
5. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
6. Hoare, T., O'Hearn, P.: Separation Logic Semantics for Communicating Processes. Electronic Notes in Theoretical Computer Science 212, 3–25 (2008)
7. Hobor, A.: Oracle Semantics. PhD thesis, Princeton University (October 2008)
8. Hurlin, C.: Automatic Parallelization and Optimization of Programs by Proof Rewriting. In: Palsberg, J., Su, Z. (eds.) Static Analysis. LNCS, vol. 5673, pp. 52–68. Springer, Heidelberg (2009)
9. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM Symposium on Principles of Programming Languages (POPL), pp. 42–54. ACM Press, New York (2006)
10. O'Hearn, P.W.: Resources, Concurrency, and Local Reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)
11. Ottoni, G.: Global Multi-Threaded Instruction Scheduling: Technique and Initial Results. PhD thesis, Princeton University (September 2008)
12. Rangan, R.: Pipelined Multithreading Transformations and Support Mechanisms. PhD thesis, Princeton University (June 2004)
13. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled software pipelining with the synchronization array. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT) (September 2004)
14. Turon, A., Wand, M.: A separation logic for the pi-calculus (2009), http://www.ccs.neu.edu/home/turon/pi-sep-logic.pdf
15. Vachharajani, N., Rangan, R., Raman, E., Bridges, M.J., Ottoni, G., August, D.I.: Speculative Decoupled Software Pipelining. In: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT) (September 2007)

Automatic Abstraction for Intervals Using Boolean Formulae

Jörg Brauer[1] and Andy King[2]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Portcullis Computer Security, Pinner, UK

Abstract. Traditionally, transfer functions have been manually designed for each operation in a program. Recently, however, there has been growing interest in computing transfer functions, motivated by the desire to reason about sequences of operations that constitute basic blocks. This paper focuses on deriving transfer functions for intervals — possibly the most widely used numeric domain — and shows how they can be computed from Boolean formulae which are derived through bit-blasting. This approach is entirely automatic, avoids complicated elimination algorithms, and provides a systematic way of handling wrap-arounds (integer overflows and underflows) which arise in machine arithmetic.

1 Introduction

The key idea in abstract interpretation [6] is to simulate the execution of each concrete operation $g : C \to C$ in a program with an abstract analogue $f : D \to D$ where C and D are domains of concrete values and descriptions. Each abstract operation f is designed to faithfully model its concrete counterpart g in the sense that if $d \in D$ describes a concrete value $c \in C$, sometimes written relationally as $d \propto c$ [17], then the result of applying g to c is described by the action of applying f to d, that is, $f(d) \propto g(c)$. Even for a fixed set of abstractions, there are typically many ways of designing the abstract operations. Ideally the abstract operations should compute abstractions that are as descriptive, that is, as accurate as possible, though there is usually interplay with accuracy and complexity, which is one reason why the literature is so rich. Normally the abstract operations are manually designed up front, prior to the analysis itself, but there are distinct advantages in synthesising the abstract operations from their concrete versions as part of the analysis itself, in a fully automatic way.

1.1 The Drive for Automatic Abstraction

One reason for automation stems from operations that arise in sequences that are known as blocks. Suppose that such a sequence is formed of n concrete operations $g_1, g_2, \ldots, g_n$, and each operation g_i has its own abstract counterpart f_i, henceforth referred to as its transfer function. Suppose too that the input to the sequence $c \in C$ is described by an input abstraction $d \in D$, that is, $d \propto c$.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 167–183, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Then the result of applying the n concrete operations to the input (one after another) is described by applying the composition of the n transfer functions to the abstract input, that is, $f_n(\ldots f_2(f_1(d))) \propto g_n(\ldots g_2(g_1(c)))$. However, a more accurate result can be obtained by deriving a single transfer function f for the block $g_n \circ \ldots \circ g_2 \circ g_1$ as a whole, designed so that $f(d) \propto g_n(\ldots g_2(g_1(c)))$. The value of this approach has been demonstrated for linear congruences [11] in the context of verifying bit-twiddling code [14]. Since blocks are program dependent, such an approach relies on automation rather than human intervention.

Another compelling reason for automation is the complexity of the concrete operations themselves. Even a simple concrete operation, such as increment by one, is complicated by the finite nature of computer arithmetic: if increment is applied to the largest integer that can be stored in a word, then the result is the smallest integer that is representable. As the transfer function needs to faithfully simulate concrete increment, then the corner case inevitably manifests itself (if not in the transfer function itself then elsewhere [29]). The problem of deriving transfer functions for low-level instructions, such as those of the x86, is particularly acute [2] since these operations not only update registers and memory locations, but also side effect status flags. Automatic abstraction offers a way to potentially tame this complexity.

1.2 Specifying Extreme Values with Universal Quantifiers

Monniaux [20] recently addressed the vexing question of automatic abstraction by focussing on template domains [28] which include, most notably, intervals [7]. He showed that if the concrete operations are specified as piecewise linear functions, then it is possible to derive transfer functions for blocks. The transfer functions relate the values of variables on entry to a block to their values on exit. To illustrate, suppose the variables x, y and z occur in a block and consider the maximum value of x on exit from the block. Monniaux shows how quantification can be used to specify the maximal output value of x in terms of the extreme values that x, y and z can take on entry to the block. The specification states that: the maximal output value of x is an upper bound on all the output values of x that are feasible for the values of x, y and z that fall within their input ranges. It also asserts that: the maximal output value of x is smaller than any other upper bound on the output value of x. These requirements are naturally formulated with universal quantification. Universal quantifier elimination is then used to find a direct linear relationship between the maximal value of x on exit and the ranges of x, y and z on entry; it is direct in that intermediate variables that occur in the specification are removed. This construction is ingenious but no polynomial elimination algorithm is known for piecewise systems, or is ever likely to exist [3]. Indeed, this computational bottleneck remains a problem [21].

1.3 Finessing Universal Quantifiers with Boolean Formulae

This paper suggests that as an alternative to operating over piecewise linear systems one can instead express the semantics of a basic block with a Boolean

formula; an idea that is familar in model checking where it is colloquially referred to as bit-blasting. Since Boolean formulae are more expressive than piecewise linear formulae, one would expect universal quantifier elimination to be just as difficult for Boolean formulae (or even harder since they are discrete). However, this is not so. To illustrate, consider $\forall x.f$ where $f = (x \vee \neg y) \wedge (\neg x \vee y \vee \neg z)$. Then $\forall x.f = f[x \mapsto 0] \wedge f[x \mapsto 1] = (\neg y) \wedge (y \vee \neg z)$. Observe that $\forall x.f$ can be obtained directly from f by removing the x and $\neg x$ literals from all the clauses of f. This is no coincidence and holds for any formula f presented in CNF not containing a vacuous clause that includes both x and $\neg x$ [15]. This suggests the following four step method for automatically deriving transfer functions for intervals (and related domains [18,19]): First, use bit-vector logic to represent the semantics of a block as a single CNF formula f_{block} (an excellent tutorial on flattening bit-vector logic into propositional logic is given in [15, Chap. 6]). Thus each n-bit integer variable is represented as a separate vector of n propositional variables. Second, apply the specification of Monniaux [20, Sect. 3.2] to express the maximal value (or conversely the minimal value) of an output bit-vector in terms of the ranges on the input bit-vectors. This gives a propositional formula f_{spec} which is essentially f_{block} augmented with universal quantifiers. Third, the universal quantifiers are removed from f_{spec} to obtain f_{simp} – a simplification of f_{spec}. Thus although universal qualification is a hinderance for linear piecewise functions, it is actually helps in a propositional formulation. Of course, f_{simp} is just a formula and does not prescribe how to compute a transfer function. However, a transfer function can be extracted from f_{simp} by abstracting f_{simp} with linear affine equations [13] which directly relate the output ranges to the input ranges. This fourth step (which is analogous to that proposed for abstracting formulae with congruences [14]) is the final step in the construction. Overall, this new approach to computing transfer functions confers the following advantages:

- it is amenable to instructions whose semantics is presented as Boolean formulae. The force of this is that propositional encodings are readily available for instructions, due to the rise in popularity of SAT-based model checking. Moreover, it is not obvious how to express the semantics of some (bit-level) instructions with piecewise linear functions;
- it avoids the computational problems associated with eliminating variables from piecewise linear systems;
- it distills transfer functions from Boolean formulae that are action systems of guarded updates. The guards are systems of octagonal constraints [19]. A guard tests whether a particular behaviour can arise, for example, whether an operation wraps, and the update revises the ranges accordingly. One guarded update might be applicable when an operation underflows and another when it overflows, thus a transfer function is a system of guarded updates. The updates are expressed with affine equations that specify how the extreme values of variables at the end of the block relate to their extreme values on entry into the block. The guards that are derived are optimal (for the class of octagons) as are the update operations (for the class of affine equalities). Once derived, a transfer function is evaluated using linear programming.

2 Worked Examples

The ethos of our approach is to express the semantics of a block in the computational domain of Boolean formulae. This concrete domain is rich enough to allow the extreme values (ranges) of variables to be specified in a way that is analogous to that of Monniaux [20]. However, in contrast to Monniaux, universal quantifier elimination is performed in the concrete setting, which is attractive computationally. Abstraction is then applied to synthesise guarded updates from quantifier-free formulae. Thus, the approach of Monniaux is abstraction then elimination, whereas ours is elimination then abstraction. We illustrate the power of this transposition by deriving transfer functions for some illustrative blocks of ATmega16 8-bit microcontroller instructions [1].

2.1 Deriving a Transfer Function for a Block

Consider deriving a transfer function for the sequence of instructions `EOR R0 R1`; `EOR R1 R0`; `EOR R0 R1` that constitutes a block. An instruction `EOR R0 R1` stores the exclusive-or of registers `R0` and `R1` in `R0`. The operands are unsigned. To specify the semantics of the block, let $\boldsymbol{r0}$ and $\boldsymbol{r1}$ denote 8-bit vectors of propositional variables that will be used to represent the symbolic initial values of `R0` and `R1`, and likewise let $\boldsymbol{r0'}$ and $\boldsymbol{r1'}$ be bit-vectors of propositional variables that denote their final values. Furthermore, let $\boldsymbol{x}[i]$ denote the ith element of the bit-vector $\boldsymbol{x}$ where $\boldsymbol{x}[0]$ is the low bit. By introducing a bit-vector $\boldsymbol{y}$ to denote the intermediate value of `R0` (which is akin to applying static single assigment) the semantics of the block can be stated propositionally as:

$$\begin{aligned}\varphi(\boldsymbol{y}) = (\wedge_{i=0}^{7}\boldsymbol{y}[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \boldsymbol{r1}[i]) \;\wedge \\ (\wedge_{i=0}^{7}\boldsymbol{r1'}[i] \leftrightarrow \boldsymbol{y}[i] \oplus \boldsymbol{r1}[i]) \wedge (\wedge_{i=0}^{7}\boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{y}[i] \oplus \boldsymbol{r1'}[i])\end{aligned}$$

where $\oplus$ denotes exclusive-or. Such formulae can be derived algorithmically by composing formulae [5,14] – one formula for each instruction in the sequence.

The formula $\varphi(\boldsymbol{y})$ specifies the relationship between the inputs $\boldsymbol{r0}$ and $\boldsymbol{r1}$ and the outputs $\boldsymbol{r0'}$ and $\boldsymbol{r1'}$, but not a relationship between their ranges. This has to be derived. To do so, let the bit-vectors $\boldsymbol{r0}_\ell$ and $\boldsymbol{r0}_u$ (resp. $\boldsymbol{r1}_\ell$ and $\boldsymbol{r1}_u$) denote the minimal and maximal values for $\boldsymbol{r0}$ (resp. $\boldsymbol{r1}$). To express these ranges in propositional logic, define the formula:

$$\boldsymbol{x} \leq \boldsymbol{y} = (\boldsymbol{x}[7] \wedge \neg\boldsymbol{y}[7]) \vee (\vee_{j=0}^{6}(\neg\boldsymbol{x}[j] \wedge \boldsymbol{y}[j] \wedge (\wedge_{k=j+1}^{7}\boldsymbol{x}[k] \leftrightarrow \boldsymbol{y}[k])))$$

and for abbreviation let $\boldsymbol{x} \leq \boldsymbol{y} \leq \boldsymbol{z} = (\boldsymbol{x} \leq \boldsymbol{y}) \wedge (\boldsymbol{y} \leq \boldsymbol{z})$. Moreover, let $\phi = (\boldsymbol{r0}_\ell \leq \boldsymbol{r0} \leq \boldsymbol{r0}_u) \wedge (\boldsymbol{r1}_\ell \leq \boldsymbol{r1} \leq \boldsymbol{r1}_u)$ to express the requirement that $\boldsymbol{r0}$ and $\boldsymbol{r1}$ are confined to their ranges. With $\boldsymbol{r0}$ and $\boldsymbol{r1}$ in range, let $\boldsymbol{r0}_\ell^\star$ and $\boldsymbol{r0}_u^\star$ denote the resulting extreme values for $\boldsymbol{r0'}$. This amounts to requiring that, firstly, $\boldsymbol{r0}_\ell^\star$ and $\boldsymbol{r0}_u^\star$ are respectively lower and upper bounds on the range of $\boldsymbol{r0'}$. Secondly, any other lower and upper bounds on $\boldsymbol{r0'}$, say, $\boldsymbol{r0}'_\ell$ and $\boldsymbol{r0}'_u$, are respectively less or equal to and greater or equal to $\boldsymbol{r0}_\ell^\star$ and $\boldsymbol{r0}_u^\star$. Analogous

requirements hold for the extreme values $\boldsymbol{r1}_\ell^\star$ and $\boldsymbol{r1}_u^\star$ of $\boldsymbol{r1}'$. We impose the first requirement with the formula $\theta(\boldsymbol{y}) = \forall \boldsymbol{r0} : \forall \boldsymbol{r1} : \forall \boldsymbol{r0}' : \forall \boldsymbol{r1}' : \theta'(\boldsymbol{y})$ where:

$$\theta'(\boldsymbol{y}) = (\phi \wedge \varphi(\boldsymbol{y})) \Rightarrow (\boldsymbol{r0}_\ell^\star \leq \boldsymbol{r0}' \leq \boldsymbol{r0}_u^\star \wedge \boldsymbol{r1}_\ell^\star \leq \boldsymbol{r1}' \leq \boldsymbol{r1}_u^\star)$$

A quantifier-free version of $\theta(\boldsymbol{y})$ is then obtained by putting $\theta'(\boldsymbol{y})$ into CNF using standard transformations [25]. This introduces fresh variables, denoted $\boldsymbol{y}'$, and thus we write the CNF formula as $\theta''(\boldsymbol{y}, \boldsymbol{y}')$. The intermediate variables of $\boldsymbol{y}$ and $\boldsymbol{y}'$ (which are existentially quantified) are then removed by repeatedly applying resolution [15]. Those literals that involve variables in $\boldsymbol{r0}, \boldsymbol{r1}, \boldsymbol{r0}'$ and $\boldsymbol{r1}'$ are then simply struck out to obtain the desired quantifier-free model $\theta(\boldsymbol{y}, \boldsymbol{y}')$.

The second requirement is enforced by introducing other lower and upper bounds on $\boldsymbol{r0}'$ and $\boldsymbol{r1}'$, namely, $\boldsymbol{r0}_\ell'$ and $\boldsymbol{r0}_u'$, and $\boldsymbol{r1}_\ell'$ and $\boldsymbol{r1}_u'$. The requirement is formally stipulated as:

$$\psi(\boldsymbol{z}) = \forall \boldsymbol{r0}_\ell' : \forall \boldsymbol{r0}_u' : \forall \boldsymbol{r1}_\ell' : \forall \boldsymbol{r1}_u' : \forall \boldsymbol{r0} : \forall \boldsymbol{r1} : \forall \boldsymbol{r0}' : \forall \boldsymbol{r1}' : \psi'(\boldsymbol{z})$$

where:

$$\psi'(\boldsymbol{z}) = ((\phi \wedge \varphi(\boldsymbol{z})) \Rightarrow (\boldsymbol{r0}_\ell' \leq \boldsymbol{r0}' \leq \boldsymbol{r0}_u' \wedge \boldsymbol{r1}_\ell' \leq \boldsymbol{r1}' \leq \boldsymbol{r1}_u')) \Rightarrow \kappa$$

and $\kappa = \boldsymbol{r0}_\ell' \leq \boldsymbol{r0}_\ell^\star \wedge \boldsymbol{r0}_u^\star \leq \boldsymbol{r0}_u' \wedge \boldsymbol{r1}_\ell' \leq \boldsymbol{r1}_\ell^\star \wedge \boldsymbol{r1}_u^\star \leq \boldsymbol{r1}_u'$. As before, we derive a quantifier-free version of $\psi(\boldsymbol{z})$, namely $\psi(\boldsymbol{z}, \boldsymbol{z}')$, where $\boldsymbol{z}'$ are the fresh variables introduced in CNF conversion. To avoid accidental variable coupling between $\theta(\boldsymbol{y}, \boldsymbol{y}')$ and $\psi(\boldsymbol{z}, \boldsymbol{z}')$ we apply renaming (if necessary) to ensure that $(var(\boldsymbol{y}) \cup var(\boldsymbol{y}')) \cap (var(\boldsymbol{z}) \cup var(\boldsymbol{z}')) = \emptyset$ where $var(o)$ denotes the set of propositional variables in the object o.

Finally, the relationship between the bounds $\boldsymbol{r0}_\ell$, $\boldsymbol{r0}_u$, $\boldsymbol{r1}_\ell$ and $\boldsymbol{r1}_u$ and the extrema $\boldsymbol{r0}_\ell^\star$, $\boldsymbol{r0}_u^\star$, $\boldsymbol{r1}_\ell^\star$ and $\boldsymbol{r1}_u^\star$, is specified with the conjunction:

$$f_{\text{simp}} = \theta(\boldsymbol{y}, \boldsymbol{y}') \wedge \psi(\boldsymbol{z}, \boldsymbol{z}')$$

The formula f_{simp} is free from universal quantifiers and, moreover, we can abstract it using affine equations [13] to discover linear relationships between the variables of $S = \{\boldsymbol{r0}_\ell^\star, \boldsymbol{r0}_u^\star, \boldsymbol{r1}_\ell^\star, \boldsymbol{r1}_u^\star, \boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r1}_\ell, \boldsymbol{r1}_u\}$ as desired. We regard this abstraction operation, denoted $\alpha_{\text{aff}}(f_{\text{simp}}, S)$, as a blackbox and defer presentation of the details to the following section. However, to state the outcome, let $\langle \boldsymbol{x} \rangle = \sum_{i=0}^{7} 2^i \boldsymbol{x}[i]$ where $\boldsymbol{x}$ is an 8-bit vector of propositional variables. Then

$$\alpha_{\text{aff}}(f_{\text{simp}}, S) = \begin{cases} \langle \boldsymbol{r0}_\ell^\star \rangle = \langle \boldsymbol{r1}_\ell \rangle \wedge \langle \boldsymbol{r0}_u^\star \rangle = \langle \boldsymbol{r1}_u \rangle \wedge \\ \langle \boldsymbol{r1}_\ell^\star \rangle = \langle \boldsymbol{r0}_\ell \rangle \wedge \langle \boldsymbol{r1}_u^\star \rangle = \langle \boldsymbol{r0}_u \rangle \end{cases}$$

This shows that the ranges of `R0` and `R1` are swapped. Indeed, the sequence of `EOR` instructions is a common idiom for exchanging the contents of two registers without employing a third. The resulting transfer function can be realised with four updates. For this example, it is difficult to see how range analysis can be usefully performed without deriving a transfer function at this level of granularity. Furthermore, it is not clear how such a transfer function could be derived from a system of piecewise linear functions [20] since such systems cannot express exclusive-or constraints.

2.2 Deriving a Transfer Function for an Operation with Many Modes

As a second example, consider computing a transfer function for the single operation `ADD R0 R1` which calculates the sum of `R0` and `R1` and stores the result in `R0`. We assume that the operands are signed and to interpret the value of such a vector let $\langle\langle \boldsymbol{x} \rangle\rangle = (\sum_{i=0}^{6} 2^i \boldsymbol{x}[i]) - 2^7 \boldsymbol{x}[7]$ where $\boldsymbol{x}[7]$ is read as the sign bit. The `ADD R0 R1` instruction is interesting because it is an exemplar of an instruction that can operate in one of three modes: it overflows (the sum exceeds 127); it underflows (the sum is strictly less than -128); or it neither overflows nor underflows (it is exact). The semantics in these respective modes, can be expressed with three Boolean formulae that are defined as follows:

$$\begin{array}{l} \varphi_O(\boldsymbol{c}) = \varphi(\boldsymbol{c}) \wedge (\neg \boldsymbol{r0}[7] \wedge \neg \boldsymbol{r1}[7] \wedge \boldsymbol{r0'}[7]) \\ \varphi_U(\boldsymbol{c}) = \varphi(\boldsymbol{c}) \wedge (\boldsymbol{r0}[7] \wedge \boldsymbol{r1}[7] \wedge \neg \boldsymbol{r0'}[7]) \\ \varphi_E(\boldsymbol{c}) = \varphi(\boldsymbol{c}) \wedge (\neg \boldsymbol{r0}[7] \vee \neg \boldsymbol{r1}[7] \vee \boldsymbol{r0'}[7]) \wedge (\boldsymbol{r0}[7] \vee \boldsymbol{r1}[7] \vee \neg \boldsymbol{r0'}[7]) \end{array}$$

where $\boldsymbol{c}$ is a bit-vector that represents the intermediate carry bits and:

$$\begin{array}{l} \varphi(\boldsymbol{c}) = \left(\bigwedge_{i=0}^{7} \boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \boldsymbol{r1}[i] \oplus \boldsymbol{c}[i] \right) \wedge \\ \qquad \neg \boldsymbol{c}[0] \wedge \left(\bigwedge_{i=0}^{6} \boldsymbol{c}[i+1] \leftrightarrow (\boldsymbol{r0}[i] \wedge \boldsymbol{r1}[i]) \vee (\boldsymbol{r0}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{r1}[i] \wedge \boldsymbol{c}[i]) \right) \end{array}$$

It is important to appreciate that the formulae that model an instruction need to be prescribed just once for each instruction. Thus, the complexity is barely more than that of bit-blasting.

The transfer functions for multi-modal operations are formulated as action systems of guarded updates. Given a system of input intervals, which defines a hypercube, the guards test whether an update is applicable and, if so, the corresponding update is applied. The update maps the input intervals to the resulting output intervals. An update is applicable if interval and guard constraints are simultaneously satisfiable. The guards are formulated as octagonal constraints [19] over the inputs to the (trivial) block, namely, $S' = \{\boldsymbol{r0}, \boldsymbol{r1}\}$. Guards are derived with an abstraction operator, denoted $\alpha_{\text{oct}}(\varphi_i(\boldsymbol{c}), S')$, which discovers the octagonal inequalities that hold in the formula $\varphi_i(\boldsymbol{c})$ between the variables of S'. This abstraction can be calculated automatically, though for reasons of continuity we defer the details until the following section. Applying this abstraction to the three $\varphi_i(\boldsymbol{c})$ formulae yields the guards:

$$\begin{array}{l} \alpha_{\text{oct}}(\varphi_O(\boldsymbol{c}), S') = \begin{cases} 128 \leq \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq 254 \wedge \\ 1 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq 127 \qquad \wedge 1 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127 \end{cases} \\ \alpha_{\text{oct}}(\varphi_U(\boldsymbol{c}), S') = \begin{cases} -256 \leq \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq -129 \wedge \\ -128 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq -1 \qquad \wedge -128 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq -1 \end{cases} \\ \alpha_{\text{oct}}(\varphi_E(\boldsymbol{c}), S') = \begin{cases} -128 \leq \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127 \wedge \\ -128 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq 127 \qquad \wedge -128 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127 \end{cases} \end{array}$$

Guards are computed using perfect integers (the detail of which is explained in the following section) rather than modulo 256. Hence the linear inequality

$\langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq 254$ which follows from the positivity requirements on $\langle\langle \boldsymbol{r0} \rangle\rangle$ and $\langle\langle \boldsymbol{r1} \rangle\rangle$, namely, $1 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq 127$ and $1 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127$.

An affine update is computed for each mode $\varphi_i(\boldsymbol{c})$ from the Boolean formula $f_{i,\text{simp}} = \theta_i(\boldsymbol{c}, \boldsymbol{c}') \wedge \psi_i(\boldsymbol{d}, \boldsymbol{d}')$ where $\theta_i(\boldsymbol{c}, \boldsymbol{c}')$ and $\psi_i(\boldsymbol{d}, \boldsymbol{d}')$ are quantifier-free formulae derived from $\varphi_i(\boldsymbol{c})$ in an analogous way to before. As before, we desire affine relationships over $S = \{\boldsymbol{r0}^\star_\ell, \boldsymbol{r0}^\star_u, \boldsymbol{r1}^\star_\ell, \boldsymbol{r1}^\star_u, \boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r1}_\ell, \boldsymbol{r1}_u\}$, hence we calculate $\alpha_{\text{aff}}(f_{i,\text{simp}}, S)$ which yields:

$$\alpha_{\text{aff}}(f_{O,\text{simp}}, S) = \begin{cases} \langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle = \langle\langle \boldsymbol{r0}_\ell \rangle\rangle + \langle\langle \boldsymbol{r1}_\ell \rangle\rangle - 256 \wedge \\ \langle\langle \boldsymbol{r0}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r0}_u \rangle\rangle + \langle\langle \boldsymbol{r1}_u \rangle\rangle - 256 \wedge \\ \langle\langle \boldsymbol{r1}^\star_\ell \rangle\rangle = \langle\langle \boldsymbol{r1}_\ell \rangle\rangle \wedge \langle\langle \boldsymbol{r1}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r1}_u \rangle\rangle \end{cases}$$

$$\alpha_{\text{aff}}(f_{U,\text{simp}}, S) = \begin{cases} \langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle = \langle\langle \boldsymbol{r0}_\ell \rangle\rangle + \langle\langle \boldsymbol{r1}_\ell \rangle\rangle + 256 \wedge \\ \langle\langle \boldsymbol{r0}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r0}_u \rangle\rangle + \langle\langle \boldsymbol{r1}_u \rangle\rangle + 256 \wedge \\ \langle\langle \boldsymbol{r1}^\star_\ell \rangle\rangle = \langle\langle \boldsymbol{r1}_\ell \rangle\rangle \wedge \langle\langle \boldsymbol{r1}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r1}_u \rangle\rangle \end{cases}$$

$$\alpha_{\text{aff}}(f_{E,\text{simp}}, S) = \begin{cases} \langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle = \langle\langle \boldsymbol{r0}_\ell \rangle\rangle + \langle\langle \boldsymbol{r1}_\ell \rangle\rangle \wedge \\ \langle\langle \boldsymbol{r0}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r0}_u \rangle\rangle + \langle\langle \boldsymbol{r1}_u \rangle\rangle \wedge \\ \langle\langle \boldsymbol{r1}^\star_\ell \rangle\rangle = \langle\langle \boldsymbol{r1}_\ell \rangle\rangle \wedge \langle\langle \boldsymbol{r1}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r1}_u \rangle\rangle \end{cases}$$

When coupled with the guards, this gives an action system of three guarded updates reflecting the three distinct modes of operation.

To illustrate an application of the derived transfers function, suppose $\langle\langle \boldsymbol{r0} \rangle\rangle$ and $\langle\langle \boldsymbol{r1} \rangle\rangle$ are clamped to fall within given input range. Moreover, suppose the range is expressed as the following system of inequalities:

$$r = \begin{cases} \langle\langle \boldsymbol{r0}_\ell \rangle\rangle = -2 \wedge \langle\langle \boldsymbol{r0}_u \rangle\rangle = 5 \quad \wedge \langle\langle \boldsymbol{r0}_\ell \rangle\rangle \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq \langle\langle \boldsymbol{r0}_u \rangle\rangle \\ \langle\langle \boldsymbol{r1}_\ell \rangle\rangle = 1 \quad \wedge \langle\langle \boldsymbol{r1}_u \rangle\rangle = 126 \wedge \langle\langle \boldsymbol{r1}_\ell \rangle\rangle \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq \langle\langle \boldsymbol{r1}_u \rangle\rangle \end{cases}$$

In addition to bound the extreme output values let:

$$r' = \begin{cases} -128 \leq \langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle \leq 127 \wedge -128 \leq \langle\langle \boldsymbol{r0}^\star_u \rangle\rangle \leq 127 \wedge \\ -128 \leq \langle\langle \boldsymbol{r1}^\star_\ell \rangle\rangle \leq 127 \wedge -128 \leq \langle\langle \boldsymbol{r1}^\star_u \rangle\rangle \leq 127 \end{cases}$$

Now consider the overflow mode. Observe that the output value of $\langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle$ can be found by solving a linear programming problem that minimises $\langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle$ subject to the linear system $r \wedge \alpha_{\text{oct}}(\varphi_O(\boldsymbol{c}), S') \wedge \alpha_{\text{aff}}(f_{O,\text{simp}}, S) \wedge r'$. This gives $\langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle = -128$. By solving three more linear programming problems we can likewise deduce $\langle\langle \boldsymbol{r0}^\star_u \rangle\rangle = -125$, $\langle\langle \boldsymbol{r1}^\star_\ell \rangle\rangle = 1$ and $\langle\langle \boldsymbol{r1}^\star_u \rangle\rangle = 126$. When repeating this process for the underflow mode, we find that the system $r \wedge \alpha_{\text{oct}}(\varphi_U(\boldsymbol{c}), S') \wedge \alpha_{\text{aff}}(f_{U,\text{simp}}, S) \wedge r'$ is infeasible, hence this guarded update is not applicable for the given input range. However, the final mode is applicable (like the first) and gives the extrema: $\langle\langle \boldsymbol{r0}^\star_\ell \rangle\rangle = -1$, $\langle\langle \boldsymbol{r0}^\star_u \rangle\rangle = 127$, $\langle\langle \boldsymbol{r1}^\star_\ell \rangle\rangle = 1$ and $\langle\langle \boldsymbol{r1}^\star_u \rangle\rangle = 126$. More generally, evaluating a system of m guarded updates for a block that involves n variables will require at most $2mn$ linear programs to be solved.

The overall result is obtained by merging the results from different modes, and there is no reason why power sets could not be deployed to summarise the final value of $\langle\langle \boldsymbol{r0} \rangle\rangle$ as $[-128, -125] \cup [-1, 127]$ (with the understanding that adjacent and overlapping intervals are merged for compactness).

2.3 Deriving a Transfer Function for a Block with Many Modes

Consider the following non-trivial sequence of instructions:

1: `INC R0;`	2: `MOV R1, R0;`	3: `LSL R1;`
4: `SBC R1, R1;`	5: `EOR R0, R1;`	6: `SUB R0, R1;`

`INC R0` increments `R0`; `MOV R1, R0` copies the contents of `R0` into `R1`; `LSL R1` leftshifts `R1` by one bit position setting the carry to the sign; `SBC R1, R1` subtracts `R1`, summed with the carry, from `R1` and stores the result in `R1`; and `SUB R0, R1` subtracts `R1` from `R0` without considering the carry. (The net effect of instructions 3 and 4 is to set all the lower bits of `R1` to its sign bit, so that `R1` contains either 0 or -1.)

Analogous to before, the sequence is bit-blasted using additional bit-vectors $\boldsymbol{w}$, $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ to represent intermediate values of registers: $\boldsymbol{w}$ for the value of `R0` immediately after instruction 1 (and the value of `R1` after instruction 2); $\boldsymbol{x}$ for the value of `R0` after instruction 5; $\boldsymbol{y}$ for the negation of `R1` after instruction 5 which is then used in the following subtraction; and $\boldsymbol{z}$ for the carry bits that are also used in subtract. With some simplification (needed to make the presentation accessible) the semantics of the block can be expressed as:

$$\varphi(\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \begin{cases} \left(\wedge_{i=0}^{7}\boldsymbol{w}[i] \leftrightarrow (\boldsymbol{r0}[i] \oplus \wedge_{j=0}^{i-1}\boldsymbol{r0}[j])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{r1'}[i] \leftrightarrow \boldsymbol{w}[7]\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{x}[i] \leftrightarrow (\boldsymbol{w}[i] \oplus \boldsymbol{r1'}[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{y}[i] \leftrightarrow (\neg\boldsymbol{r1'}[i] \oplus \wedge_{j=0}^{i-1}\neg\boldsymbol{r1'}[j])\right) & \wedge \\ (\neg\boldsymbol{z}[0]) & \wedge \\ \left(\wedge_{i=0}^{6}\boldsymbol{z}[i+1] \leftrightarrow (\boldsymbol{x}[i] \wedge \boldsymbol{y}[i]) \vee (\boldsymbol{x}[i] \wedge \boldsymbol{z}[i]) \vee (\boldsymbol{y}[i] \wedge \boldsymbol{z}[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7}\boldsymbol{r0'}[i] \leftrightarrow \boldsymbol{x}[i] \oplus \boldsymbol{y}[i] \oplus \boldsymbol{z}[i]\right) & \end{cases}$$

For brevity, let vector $\boldsymbol{u}$ denote the concatenation of the vectors $\boldsymbol{w}$, $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ so as to write $\varphi(\boldsymbol{u})$ for $\varphi(\boldsymbol{w},\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$.

For the increment, there are two modes of operation depending on whether it overflows or not. These can be expressed by: $\mu_O = (\neg\boldsymbol{r0}[7]) \wedge (\wedge_{i=0}^{6}\boldsymbol{r0}[i])$ and $\mu_E = \neg\mu_O$ respectively. For the leftshift, there are two modes depending on whether it overflows or not, as defined by the formulae $\eta_O = \boldsymbol{w}[7]$ and $\eta_E = \neg\eta_O$ respectively. For subtraction, there are again three modes according to whether it overflows or underflows or does neither. These modes can be expressed as: $\nu_U = (\neg\boldsymbol{r0'}[7] \wedge \boldsymbol{x}[7] \wedge \boldsymbol{y}[7])$, $\nu_O = (\boldsymbol{r0'}[7] \wedge \neg\boldsymbol{x}[7] \wedge \neg\boldsymbol{y}[7])$ and $\nu_E = (\neg\nu_U) \wedge (\neg\nu_O)$. All other instructions in the block are unimodal; indeed they neither overflow nor underflow [1]. This gives twelve different mode combinations overall. However, the following formulae are unsatisfiable:

$$\begin{array}{lll} \mu_O \wedge \eta_O \wedge \nu_O \wedge \varphi(\boldsymbol{u}) & \mu_O \wedge \eta_O \wedge \nu_E \wedge \varphi(\boldsymbol{u}) & \mu_O \wedge \eta_E \wedge \nu_U \wedge \varphi(\boldsymbol{u}) \\ \mu_O \wedge \eta_E \wedge \nu_O \wedge \varphi(\boldsymbol{u}) & \mu_O \wedge \eta_E \wedge \nu_E \wedge \varphi(\boldsymbol{u}) & \mu_E \wedge \eta_O \wedge \nu_U \wedge \varphi(\boldsymbol{u}) \\ \mu_E \wedge \eta_O \wedge \nu_O \wedge \varphi(\boldsymbol{u}) & \mu_E \wedge \eta_E \wedge \nu_U \wedge \varphi(\boldsymbol{u}) & \mu_E \wedge \eta_E \wedge \nu_O \wedge \varphi(\boldsymbol{u}) \end{array}$$

indicating these mode combinations are not feasible. Henceforth, only three combinations needed to be considered when synthesising the action system.

Hence let $\varphi_1(\boldsymbol{u}) = \mu_O \wedge \eta_O \wedge \nu_U \wedge \varphi(\boldsymbol{u})$, $\varphi_2(\boldsymbol{u}) = \mu_E \wedge \eta_O \wedge \nu_E \wedge \varphi(\boldsymbol{u})$ and $\varphi_3(\boldsymbol{u}) = \mu_E \wedge \eta_E \wedge \nu_E \wedge \varphi(\boldsymbol{u})$. The inputs to the block are $S' = \{\boldsymbol{r0}, \boldsymbol{r1}\}$ and calculating guards for these combinations gives:

$$\begin{aligned}
\alpha_{\text{oct}}(\varphi_1(\boldsymbol{v}), S') &= 127 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq 127 \wedge -128 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127 \\
\alpha_{\text{oct}}(\varphi_2(\boldsymbol{v}), S') &= -128 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq -2 \ \wedge -128 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127 \\
\alpha_{\text{oct}}(\varphi_3(\boldsymbol{v}), S') &= -1 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq 126 \wedge -128 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 127
\end{aligned}$$

Note that all three guards impose vacuous constraints on $\langle\langle \boldsymbol{r1} \rangle\rangle$. This is because `R1` is written before it is read (which could be inferred prior to deriving the transfer functions though this is not strictly necessary).

As before, the updates are computed for each $\varphi_i(\boldsymbol{u})$ from three formulae $f_{i,\text{simp}} = \theta_i(\boldsymbol{u}, \boldsymbol{u}') \wedge \psi_i(\boldsymbol{v}, \boldsymbol{v}')$ where $\theta_i(\boldsymbol{u}, \boldsymbol{u}')$ and $\psi_i(\boldsymbol{v}, \boldsymbol{v}')$ are quantifier-free and derived from $\varphi_i(\boldsymbol{u})$ as previously. Hence we calculate affine relationships over $S = \{\boldsymbol{r0}^\star_\ell, \boldsymbol{r0}^\star_u, \boldsymbol{r1}^\star_\ell, \boldsymbol{r1}^\star_u, \boldsymbol{r0}_\ell, \boldsymbol{r0}_u, \boldsymbol{r1}_\ell, \boldsymbol{r1}_u\}$ which yields:

$$\begin{aligned}
\alpha_{\text{aff}}(f_{1,\text{simp}}, S) &= \langle\langle \boldsymbol{r0} \rangle\rangle_\ell = 127 \qquad \wedge \langle\langle \boldsymbol{r0} \rangle\rangle_u = 127 \qquad \wedge \\
& \langle\langle \boldsymbol{r0} \rangle\rangle^\star_\ell = -128 \qquad \wedge \langle\langle \boldsymbol{r0} \rangle\rangle^\star_u = -128 \\
\alpha_{\text{aff}}(f_{2,\text{simp}}, S) &= \langle\langle \boldsymbol{r0} \rangle\rangle^\star_\ell = -\langle\langle \boldsymbol{r0} \rangle\rangle_u - 1 \wedge \langle\langle \boldsymbol{r0} \rangle\rangle^\star_u = -\langle\langle \boldsymbol{r0} \rangle\rangle_\ell - 1 \\
\alpha_{\text{aff}}(f_{3,\text{simp}}, S) &= \langle\langle \boldsymbol{r0} \rangle\rangle^\star_\ell = \langle\langle \boldsymbol{r0} \rangle\rangle_\ell + 1 \quad \wedge \langle\langle \boldsymbol{r0} \rangle\rangle^\star_u = \langle\langle \boldsymbol{r0} \rangle\rangle_u + 1
\end{aligned}$$

Note that no affine constraints are inferred for $\boldsymbol{r1}^\star_\ell$ and $\boldsymbol{r1}^\star_u$ reflecting that `R1` is used merely to store an intermediate value (though combining affine equations with congruences [14] would preserve some information pertaining to the final value of `R1`). Note how ranges are swapped for the second mode since in this circumstance `R0` is negative. From the resulting action system, it can be seen that the block overwrites `R0` with the absolute value of $(\texttt{R0} + 1)$ subject to wrap around $(128 = -128 \mod 256)$. To the best of our knowledge, no other approach can derive a useful transfer function for a block such as this.

3 Abstracting Boolean Formulae

This section shows how to construct octagonal and affine abstractions of a given Boolean formula $\varphi(\boldsymbol{v})$ defined over a set of bit-vectors $S = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ of size k and a single bit-vector $\boldsymbol{v}$ that identifies any intermediate variables. For presentational purposes, we focus on deriving abstractions that relate the values of $\langle\langle \boldsymbol{x}_1 \rangle\rangle, \ldots, \langle\langle \boldsymbol{x}_n \rangle\rangle$, though the construction for unsigned values is analogous.

3.1 Abstracting Boolean Formulae with Octagonal Inequalities

Let $\lambda, \mu \in \{-1, 0, 1\}$ and $1 \leq i \leq j \leq n$ be fixed and consider the derivation of an octagonal inequality $\lambda\langle\langle \boldsymbol{x}_i \rangle\rangle + \mu\langle\langle \boldsymbol{x}_j \rangle\rangle \leq c^\star$ where $c^\star \in \mathbb{Z}$. Since k is fixed it follows $-2^k \leq \lambda\langle\langle \boldsymbol{x}_i \rangle\rangle + \mu\langle\langle \boldsymbol{x}_j \rangle\rangle \leq 2^k$, hence the problem reduces to finding the least $-2^k \leq c^\star \leq 2^k$ such that if $\varphi(\boldsymbol{v})$ holds then $\lambda\langle\langle \boldsymbol{x}_i \rangle\rangle + \mu\langle\langle \boldsymbol{x}_j \rangle\rangle \leq c^\star$ also holds. To this end, let $\boldsymbol{c}^\star$ denote a bit-vector of size $k+2$ and suppose $\phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u})$ is a propositional encoding of $\lambda\langle\langle \boldsymbol{x}_i \rangle\rangle + \mu\langle\langle \boldsymbol{x}_j \rangle\rangle \leq \boldsymbol{c}^\star$ where the sum is calculated

to a width of $k+2$ bits and $\leq$ is encoded as in Sect. 2.1, likewise operating on vectors of size $k+2$. In this formulation, the vector $\boldsymbol{v}$ denotes the intermediate variables required for addition and negation. Since $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are k-bit this construction avoids wraps, hence $\phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v})$ holds iff $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq c^\star$ holds. Likewise, suppose that $\phi'_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}')$ is a propositional formula that holds iff $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq c'$ holds where $\boldsymbol{c}'$ is $k+2$ bit-vector distinct from $\boldsymbol{c}^\star$. Finally, let κ denote a Boolean formula that holds iff $\boldsymbol{c}^\star \leq \boldsymbol{c}'$ holds.

Single inequalities. With the formulae $\phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v})$ and $\phi'_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}')$ thus defined, we can apply universal quantification to specify the least $c^\star$ as the unique value $\langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle$ which satisfies $\theta_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u},\boldsymbol{v}) \wedge \psi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u}',\boldsymbol{v}')$ where:

$$\begin{aligned}
\theta_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u},\boldsymbol{v}) &= \forall \boldsymbol{x}_i : \forall \boldsymbol{x}_j : (\varphi(\boldsymbol{u}) \Rightarrow \phi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v})) \\
\psi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u}',\boldsymbol{v}') &= \forall \boldsymbol{x}_i : \forall \boldsymbol{x}_j : \forall \boldsymbol{c}' : ((\varphi(\boldsymbol{u}') \Rightarrow \phi'_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{v}')) \Rightarrow \kappa)
\end{aligned}$$

and $\boldsymbol{u}$ and $\boldsymbol{u}'$ are renamed apart so as to avoid cross-coupling. More generally, the octagonal abstraction of $\varphi(\boldsymbol{v})$ over the set S is given by:

$$\alpha_{\mathrm{oct}}(\varphi(\boldsymbol{v}), S) = \bigwedge \left\{ \lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq \langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle \;\middle|\; \begin{array}{ll} \exists \lambda, \mu \in \{-1,0,1\} & \wedge \\ \exists 1 \leq i \leq j \leq n & \wedge \\ \theta_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u},\boldsymbol{v}) \wedge \psi_{\lambda,\mu,\boldsymbol{x}_i,\boldsymbol{x}_j}(\boldsymbol{u}',\boldsymbol{v}') \text{ holds} & \end{array} \right\}$$

Note that in $\lambda\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle + \mu\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \leq \langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle$ the symbols $\langle\!\langle \boldsymbol{x}_i \rangle\!\rangle$ and $\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle$ denote variables whereas $\langle\!\langle \boldsymbol{c}^\star \rangle\!\rangle$ is a value that is fixed by the formula $\theta_{\lambda,\mu,i,j}(\boldsymbol{u},\boldsymbol{v}) \wedge \psi_{\lambda,\mu,i,j}(\boldsymbol{u}',\boldsymbol{v}')$. Of course, as previously explained, quantifier-free versions of $\theta_{\lambda,\mu,i,j}(\boldsymbol{u},\boldsymbol{v})$ and $\psi_{\lambda,\mu,i,j}(\boldsymbol{u}',\boldsymbol{v}')$ can be obtained through CNF conversion and clause simplification, hence $\alpha_{\mathrm{oct}}(\varphi(\boldsymbol{v}), S)$ can be computed as well as specified. The resulting octagonal abstraction is closed (though this is not necessary in our setting).

Many inequalities. Interestingly, many inequalities can be derived in a single call to the solver. Let $\{(\boldsymbol{y}_1,\boldsymbol{z}_1),\ldots,(\boldsymbol{y}_m,\boldsymbol{z}_m)\} \subseteq S^2$ and $\{(\lambda_1,\mu_1),\ldots,(\lambda_m,\mu_m)\} \subseteq \{-1,0,1\}^2$, and consider the problem of finding the set $\{c_1^\star,\ldots,c_m^\star\} \subseteq [-2^k, 2^k-1]$ of least values such that if $\varphi(\boldsymbol{v})$ holds then $\lambda_i\langle\!\langle \boldsymbol{y}_i \rangle\!\rangle + \mu_i\langle\!\langle \boldsymbol{z}_i \rangle\!\rangle \leq c_i^\star$ holds. This problem can be formulated in an analogous way to before using bit-vectors $\boldsymbol{c}_1^\star,\ldots,\boldsymbol{c}_m^\star$ and $\boldsymbol{c}_1',\ldots,\boldsymbol{c}_m'$ all of size $k+2$. Furthermore, let κ_i be a propositional formula that holds iff $\boldsymbol{c}_i^\star \leq \boldsymbol{c}_i'$ holds. Then the problem of simultaneously finding all the $c_i^\star$ amounts to solving $\theta(\boldsymbol{u},\boldsymbol{v}_1,\ldots,\boldsymbol{v}_m) \wedge \psi(\boldsymbol{u}',\boldsymbol{v}_1',\ldots,\boldsymbol{v}_m')$ where:

$$\begin{aligned}
\theta(\boldsymbol{u},\boldsymbol{v}_1,\ldots,\boldsymbol{v}_m) &= \forall \boldsymbol{y}_1 : \forall \boldsymbol{z}_1 : \ldots : \forall \boldsymbol{y}_k : \forall \boldsymbol{z}_k : (\varphi(\boldsymbol{u}) \Rightarrow \textstyle\bigwedge_{k=1}^m \phi_{\lambda_k,\mu_k,\boldsymbol{y}_k,\boldsymbol{z}_k}(\boldsymbol{v}_k)) \\
\psi(\boldsymbol{u},\boldsymbol{v}_1',\ldots,\boldsymbol{v}_m') &= \forall \boldsymbol{y}_1 : \forall \boldsymbol{z}_1 : \ldots : \forall \boldsymbol{y}_k : \forall \boldsymbol{z}_k : \\
&\quad \forall \boldsymbol{c}_1' : \ldots : \forall \boldsymbol{c}_m' : ((\varphi(\boldsymbol{u}') \Rightarrow \textstyle\bigwedge_{k=1}^m \phi'_{\lambda_k,\mu_k,\boldsymbol{y}_k,\boldsymbol{z}_k}(\boldsymbol{v}_k')) \Rightarrow \textstyle\bigwedge_{k=1}^m \kappa_k)
\end{aligned}$$

Notice that $\theta(\boldsymbol{u},\boldsymbol{v}_1,\ldots,\boldsymbol{v}_m) \wedge \psi(\boldsymbol{u}',\boldsymbol{v}_1',\ldots,\boldsymbol{v}_m')$ involves one copy of $\varphi(\boldsymbol{u})$ and another of $\varphi(\boldsymbol{u}')$ rather than m copies of both.

Timings. The time required to bit-blast the blocks in Sect. 2 and then eliminate the universal quantifiers were essentially non-measurable. The time required to prove unsatisfiability or compute the guards for the various mode combinations varied between 0.1s and 0.6s. Octagons were derived one inequality at a time (rather than several together) using the SAT4J solver [16] on 2.6GHz MacBook Pro. Incremental SAT solving would speedup the generation of octagons, as the intermediate results used to derive one inequality could be used to infer another.

3.2 Abstracting Boolean Formulae with Affine Equalities

Affine equations [13,22] are related to congruences [11,23]; indeed the former is a special case of the latter where the modulo is 0. This suggests adapting an abstraction technique for formulae that discovers congruence relationships between the propositional variables of a given formula [14]. In our setting, the problem is different. It is that of computing an affine abstraction of a formula $\varphi(\boldsymbol{v})$ defined over a set of bit-vectors S. As before, we do not aspire to derive relationships that involve intermediate variables $\boldsymbol{v}$ and we assume each $\boldsymbol{x}_i$ is signed.

Algorithm. Figure 1 gives an algorithm for computing $\alpha_{\text{aff}}(\varphi(\boldsymbol{v}), S)$. In what follows, the n-ary vector $\boldsymbol{x}$ is defined as $\boldsymbol{x} = (\langle\!\langle \boldsymbol{x}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_n \rangle\!\rangle)$. Affine equations over S are represented with an augmented rational matrix $[A \mid \boldsymbol{b}]$ that we interpret as defining the set $\{\boldsymbol{x} \in [-2^{k-1}, 2^{k-1}]^n \mid A\boldsymbol{x} = \boldsymbol{b}\}$. The algorithm relies on a propositional encoding for an affine disequality constraint $(c_1, \ldots, c_n) \cdot \boldsymbol{x} \neq b$ where $c_1, \ldots, c_n, b \in \mathbb{Q}$. To see that such an encoding is possible assume, without loss of generality, that the disequality is integral and b is non-negative. Then rewrite the disequality as $(c_1^+, \ldots, c_n^+) \cdot \boldsymbol{x} \neq b + (c_1^-, \ldots, c_n^-) \cdot \boldsymbol{x}$ where $(c_1^+, \ldots, c_n^+), (c_1^-, \ldots, c_n^-) \in \mathbb{N}^n$ and $\mathbb{N} = \{i \in \mathbb{Z} \mid 0 \leq i\}$. Let $c^+ = \sum_{i=1}^n c_i^+$ and $c^- = \sum_{i=1}^n c_i^-$. Since each $\langle\!\langle \boldsymbol{x}_j \rangle\!\rangle \in [-2^{k-1}, 2^{k-1} - 1]$ it follows that computing the sums $(c_1^+, \ldots, c_n^+) \cdot \boldsymbol{x}$ and $b + (c_1^-, \ldots, c_n^-) \cdot \boldsymbol{x}$ with a signed $1 + \lceil \log_2(1 + \max(2^k c^+, b + 2^k c^-)) \rceil$ bit representation is sufficient to avoid wraps, allowing the disequality to be modelled exactly as a formula. In the algorithm, this formula is denoted $\phi(\boldsymbol{w})$ where $\boldsymbol{w}$ is a vector of temporary variables used for carry bits and intermediate sums.

Apart from $\phi(\boldsymbol{w})$, the abstraction algorithm is essentially the same as that proposed for congruences [14] (with a proof of correctness carrying over too). The algorithm starts with an unsatisfiable constraint $\mathbf{0} \cdot \boldsymbol{x} = 1$ which is successively relaxed by merging it with a series of affine systems that are derived by SAT (or SMT) solving. The truth assignment θ is considered to be a mapping $\theta : var(\varphi(\boldsymbol{v}) \wedge \phi(\boldsymbol{w})) \rightarrow \{0, 1\}$ which, when applied to a k-bit vector of variables such as $\boldsymbol{x}_j$, yields a binary vector. Such a binary vector can then be interpreted as a signed number to give a value in the range $[-2^{k-1}, 2^{k-1} - 1]$. This construction is applied in lines 5-8 to find a vector $(\langle\!\langle \theta(\boldsymbol{x}_1) \rangle\!\rangle, \ldots, \langle\!\langle \theta(\boldsymbol{x}_n) \rangle\!\rangle) \in [-2^{k-1}, 2^{k-1} - 1]^n$ which satisfies both the disequality $(a_1, \ldots, a_n) \cdot \boldsymbol{x} \neq b$ and the formula $\varphi(\boldsymbol{v})$.

The algorithm is formulated in terms of some auxiliary functions: row(M, i) extracts row i from the matrix M where the first row is taken to be row 1 (rather

```
(1)   function affine(φ(v), {x_1, ..., x_n})
(2)       [A | b] := [0, ..., 0 | 1];
(3)       i := 0; r := 1;
(4)       while i < r do
(5)           (a_1, ..., a_n, b) := row([A | b], r − i);
(6)           let ϕ(w) hold iff (a_1, ..., a_n) · x ≠ b holds;
(7)           if φ(v) ∧ ϕ(w) has a satisfying truth assignment θ
(8)               [A' | b'] := [A | b] ⊔_aff [Id | (⟪θ(x_1)⟫, ..., ⟪θ(x_n)⟫)^T];
(9)               [A | b] := triangular([A' | b']);
(10)              r := number_of_rows([A | b]);
(11)          else i := i + 1;
(12)      endwhile
(13)  return [A | b]
```

Fig. 1. Calculating the affine closure of the Boolean formula $\varphi(\boldsymbol{v})$

than 0); triangular(M) puts M into an upper triangular form using Gaussian elimination; and number_of_rows(M) returns the number of rows in M.

The rows of $[A \mid \boldsymbol{b}]$ are considered in reverse order. Each iteration of the loop tests whether there exists a truth assignment of $\varphi(\boldsymbol{v})$ that also satisfies the $\phi(\boldsymbol{w})$ formula constructed from row $r - i$. If not, then every model of $\varphi(\boldsymbol{v})$ satisfies the affine equality $(a_1, \ldots, a_n) \cdot \boldsymbol{x} = b$ represented by row $r - i$. Hence the equality constitutes a description of the formula. The counter i is then incremented to examine a row which, thus far, has not been considered. Conversely, if the instance is satisfiable, then the solution is represented as a matrix $[\mathsf{Id} \mid (\langle\!\langle \theta(\boldsymbol{x}_1) \rangle\!\rangle, \ldots, \langle\!\langle \theta(\boldsymbol{x}_n) \rangle\!\rangle)^T]$ which is merged with $[A|\boldsymbol{b}]$. Merge is an $O(n^3)$ operation [13] that yields a new summary $[A'|\boldsymbol{b}']$ that enlarges $[A|\boldsymbol{b}]$ with the freshly found solution. The next iteration of the loop will either relax $[A|\boldsymbol{b}]$ by finding another solution, or verify that the row now describes $\varphi(\boldsymbol{v})$. Triangular form ensures that all rows beneath the one under consideration will never be effected by the merge. At most n iterations are required since the affine systems enumerated by the algorithm constitute an ascending chain over n variables [13].

Example. Consider $\varphi(\boldsymbol{w}, \boldsymbol{x})$ which is an encoding of $\langle\!\langle \boldsymbol{z} \rangle\!\rangle = 2(\langle\!\langle \boldsymbol{v} \rangle\!\rangle + 1) + \langle\!\langle \boldsymbol{y} \rangle\!\rangle$ subject to the additional constraints that $-32 \leq \langle\!\langle \boldsymbol{v} \rangle\!\rangle \leq 31$ and $-32 \leq \langle\!\langle \boldsymbol{y} \rangle\!\rangle \leq 31$:

$$\varphi(\boldsymbol{w}, \boldsymbol{x}) = \begin{cases} (\neg \boldsymbol{w}[0]) \wedge \left(\wedge_{i=0}^{6} \boldsymbol{w}[i+1] \leftrightarrow (\boldsymbol{v}[i] \oplus \wedge_{j=0}^{i-1} \boldsymbol{v}[j])\right) & \wedge \\ (\neg \boldsymbol{x}[0]) & \wedge \\ \left(\wedge_{i=0}^{6} \boldsymbol{x}[i+1] \leftrightarrow (\boldsymbol{w}[i] \wedge \boldsymbol{x}[i]) \vee (\boldsymbol{w}[i] \wedge \boldsymbol{y}[i]) \vee (\boldsymbol{x}[i] \wedge \boldsymbol{y}[i])\right) & \wedge \\ \left(\wedge_{i=0}^{7} \boldsymbol{z}[i] \leftrightarrow \boldsymbol{w}[i] \oplus \boldsymbol{x}[i] \oplus \boldsymbol{y}[i]\right) & \wedge \\ ((\boldsymbol{v}[7] \leftrightarrow \boldsymbol{v}[6]) \wedge (\boldsymbol{v}[6] \leftrightarrow \boldsymbol{v}[5])) \wedge ((\boldsymbol{y}[7] \leftrightarrow \boldsymbol{y}[6]) \wedge (\boldsymbol{y}[6] \leftrightarrow \boldsymbol{y}[5])) & \end{cases}$$

Suppose $\boldsymbol{x}_1 = \boldsymbol{v}$, $\boldsymbol{x}_2 = \boldsymbol{y}$ and $\boldsymbol{x}_3 = \boldsymbol{z}$. The solutions that are found in each iteration are given in the left hand column. The $[A|\boldsymbol{b}]$ and $[A'|\boldsymbol{b}]$ are immediately left and right of the equality. The arrow indicates the row under consideration. The unsatisfiable case is first encountered in the final iteration. The algorithm returns $[2, 1, -1 \mid -2]$ and thus recovers $2\langle\!\langle \boldsymbol{v} \rangle\!\rangle + \langle\!\langle \boldsymbol{y} \rangle\!\rangle - \langle\!\langle \boldsymbol{z} \rangle\!\rangle = -2$ from $\varphi(\boldsymbol{v})$.

$$(0,0,2) \qquad \left[\begin{array}{ccc|c} 0 & 0 & 0 & 1 \end{array}\right] \sqcup_{\text{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \leftarrow \end{array}\right]$$

$$(-1,0,0) \qquad \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array}\right] \sqcup_{\text{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right] = \left[\begin{array}{ccc|c} 2 & 0 & -1 & -2 \\ 0 & 1 & 0 & 0 \leftarrow \end{array}\right]$$

$$(0,1,3) \qquad \left[\begin{array}{ccc|c} 2 & 0 & -1 & -2 \\ 0 & 1 & 0 & 0 \end{array}\right] \sqcup_{\text{aff}} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 3 \end{array}\right] = \left[\begin{array}{ccc|c} 2 & 1 & -1 & -2 \leftarrow \end{array}\right]$$

Timings. Computing affine abstractions for the feasible mode combinations in the examples of the previous section took no longer than 0.4s per mode. Each mode required no more than 5 SAT instances to be solved (which is considerably fewer than that required for inferring bit-level congruences [14]) with the join taking less than 5% of the overall runtime.

4 Applying Action Systems of Guarded Updates

Thus far we have derived transfer functions that are action systems of guarded updates $T = \{(g_1, u_1), \ldots, (g_m, u_m)\}$ where each guard g_i is a system of octagonal constraints over a set of bit-vectors $S' = \{\boldsymbol{x}_i \mid i \in [1,n]\}$ and each update u_i is a system of affine constraints over $S = \{\boldsymbol{x}_{i,\ell}, \boldsymbol{x}_{i,u}, \boldsymbol{x}'_{i,\ell}, \boldsymbol{x}'_{i,u} \mid i \in [1,n]\}$. In this section we show that the application of such an action system can be reduced to linear programming. To do so, we continue working with the assumption that S and S' are all k-bit and represent signed objects.

Thus consider applying T to a system of interval constraints over S' of the form $c = \bigwedge_{i=1}^{n} \ell_i \leq \langle\!\langle \boldsymbol{x}_i \rangle\!\rangle \leq u_i$ where $\{\ell_1, u_1, \ldots, \ell_n, u_n\} \subseteq [-2^{k-1}, 2^{k-1} - 1]$. In particular, consider the application of the guarded update (g_k, u_k). Suppose $g_k = \bigwedge_{i=1}^{p} \boldsymbol{\lambda}_i \cdot \boldsymbol{x} \leq d_i$ where $\boldsymbol{x}$ is the n-ary vector $\boldsymbol{x} = (\langle\!\langle \boldsymbol{x}_1 \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_n \rangle\!\rangle)$ and each n-ary coefficient vector $\boldsymbol{\lambda}_i \in \{-1, 0, 1\}^n$ has no more than two non-zero elements and $d_i \in \mathbb{Z}$. Furthermore, denote

$$\begin{array}{ll} \boldsymbol{x}_\ell = (\langle\!\langle \boldsymbol{x}_{1,\ell} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_{n,\ell} \rangle\!\rangle) & \boldsymbol{x}_u = (\langle\!\langle \boldsymbol{x}_{1,u} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}_{n,u} \rangle\!\rangle) \\ \boldsymbol{x}'_\ell = (\langle\!\langle \boldsymbol{x}'_{1,\ell} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}'_{n,\ell} \rangle\!\rangle) & \boldsymbol{x}'_u = (\langle\!\langle \boldsymbol{x}'_{1,u} \rangle\!\rangle, \ldots, \langle\!\langle \boldsymbol{x}'_{n,u} \rangle\!\rangle) \end{array}$$

Then u_k can be written as $u_k = \bigwedge_{i=1}^{q} \boldsymbol{\mu}_{i,\ell} \cdot \boldsymbol{x}_\ell + \boldsymbol{\mu}_{i,u} \cdot \boldsymbol{x}_u + \boldsymbol{\mu}'_{i,\ell} \cdot \boldsymbol{x}'_\ell + \boldsymbol{\mu}'_{i,u} \cdot \boldsymbol{x}'_u = f_i$ where each n-ary vector $\boldsymbol{\mu}_{i,\ell}, \boldsymbol{\mu}_{i,u}, \boldsymbol{\mu}'_{i,\ell}, \boldsymbol{\mu}'_{i,u} \in \mathbb{Z}^n$ and $f_i \in \mathbb{Z}$. To specify a linear program, let $\boldsymbol{e}_j$ denote the n-ary elementary vector $(0, \ldots, 0, 1, 0, \ldots, 0)$ where $j-1$ zeros precede the one and $n-j$ zeros follow it. Then the value of $\langle\!\langle \boldsymbol{x}'_{j,\ell} \rangle\!\rangle$ for any $j \in [1,n]$ can be found by minimising $\boldsymbol{e}_j \cdot \boldsymbol{x}'_\ell$ subject to:

$$P = \begin{cases} \bigwedge_{i=1}^{q} \boldsymbol{\mu}_{i,\ell} \cdot \boldsymbol{x}_\ell + \boldsymbol{\mu}_{i,u} \cdot \boldsymbol{x}_u + \boldsymbol{\mu}'_{i,\ell} \cdot \boldsymbol{x}'_\ell + \boldsymbol{\mu}'_{i,u} \cdot \boldsymbol{x}'_u = f_i & & \wedge \\ \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}_\ell = \ell_i \quad \wedge & \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}_u = u_i & \wedge \\ \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x} - \boldsymbol{e}_i \cdot \boldsymbol{x}_u \leq 0 \quad \wedge & \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}_\ell - \boldsymbol{e}_i \cdot \boldsymbol{x} \leq 0 & \wedge \\ \bigwedge_{i=1}^{n} \boldsymbol{e}_i \cdot \boldsymbol{x}'_u \leq 2^{k-1} - 1 \quad \wedge & \bigwedge_{i=1}^{n} -\boldsymbol{e}_i \cdot \boldsymbol{x}'_\ell \leq 2^{k-1} & \wedge \\ \bigwedge_{i=1}^{p} \boldsymbol{\lambda}_i \cdot \boldsymbol{x} \leq d_i & & \end{cases}$$

We let ℓ'_j denote this minima. Conversely let u'_j denote the value found by maximising $\boldsymbol{e}_j \cdot \boldsymbol{x}'_u$ subject to P. Note that although P is bounded, it is not necessarily feasible, which will be detected when solving the linear program (the first stage of two-phase simplex amounts of deciding feasibility by solving the so-called auxiliary problem [4]).

If P is feasible, the system of intervals generated for (g_k, u_k) is given by $\bigwedge_{j=1}^{n} \ell'_j \leq \langle\langle \boldsymbol{x}_j \rangle\rangle \leq u'_j$. If infeasible, the unsatisfiable constraint $\perp$ is output. Merging the systems generated by all the guarded updates (using power sets of intervals if desired) then gives the final output system of interval constraints.

Rationale. The rationale for evaluating transfer functions with linear programming is the same as that which motivated deriving transfer functions with SAT: efficient solvers are readily (even freely) available for both. Moreover, although linear solvers have suffered problems relating to floats, steady progress has been made on both speeding up exact solvers [9] and deriving safe bounds on optima [24]. Thus soundness is no longer an insurmountable problem. We do not offer timings for evaluating transfer functions, partly because the focus of this paper (reflecting that of others [14,20,26]) is on deriving them; and partly because the linear programs that arise from the examples are trivial by industrial standards.

Linear programming versus closure. Since the guards are octagonal one might wonder whether linear programming could be replaced with a closure calculation on octagons [19] that combines the interval constraints (which constitute a degenerate form of octagon) with the guard, thereby refining the interval bounds. These improved bounds could then be used to calculate the updates, without using linear programming. To demonstrate why this approach is sub-optimal, let $c = (0 \leq \langle\langle \boldsymbol{r0} \rangle\rangle \leq 1) \wedge (0 \leq \langle\langle \boldsymbol{r1} \rangle\rangle \leq 1)$ and suppose the guard is the single constraint $g = \langle\langle \boldsymbol{r0} \rangle\rangle + \langle\langle \boldsymbol{r1} \rangle\rangle \leq 1$. The closure of $c \wedge g$ would not refine the upper bounds on $\langle\langle \boldsymbol{r0} \rangle\rangle$ and $\langle\langle \boldsymbol{r1} \rangle\rangle$. Thus if these values were substituted into the update $\langle\langle \boldsymbol{r0}^\star_u \rangle\rangle = \langle\langle \boldsymbol{r0}_u \rangle\rangle + \langle\langle \boldsymbol{r1}_u \rangle\rangle$ then a maximal value of 2 would be derived for $\langle\langle \boldsymbol{r0}^\star_u \rangle\rangle$. This is safe but observe that maximising $\langle\langle \boldsymbol{r0}^\star_u \rangle\rangle$ subject to $c \wedge g$ yields the improved bound of 1, which illustrates why linear programming is preferable.

5 Related Work

The problem of computing transfer functions for numeric domains is as old as the field itself, and the seminal paper on polyhedral analysis discusses different ways to realise a transfer function for $x := x \times y$ [8, Sect. 4.2.1]. Granger [10] lamented the difficulty of handcrafting best transformers for congruences, but it took more than a decade before it was noticed that they can always be constructed for domains that satisfy the ascending chain condition [27]. The idea is to reformulate each application of a best transformer as a series of calls to a decision procedure such as a theorem prover. This differs from our work which aspires to evaluate a transfer function without a complicated decision procedure.

Contemporaneously it was observed that best transformers can be computed for intervals using BDDs [26]. The authors observe that if $g : [0, 2^8 - 1] \rightarrow$

$[0, 2^8 - 1]$ is a unary operation on an unsigned byte, then its best transformer $f : D \to D$ on $D = \{\emptyset\} \cup \{[\ell, u] \mid 0 \leq \ell \leq u < 2^8\}$ can be defined through interval subdivision. If $\ell = u$ then $f([\ell, u]) = g(\ell)$ whereas if $\ell < u$ then $f([\ell, u]) = f([\ell, m-1]) \sqcup f([m, u])$ where $m = \lfloor u/2^n \rfloor 2^n$ and $n = \lfloor \log_2(u-\ell+1) \rfloor$. Binary operations can likewise be decomposed. The 8-bit inputs, ℓ and u, can be represented as 8-bit vectors, as can the 8-bit outputs, so as to represent f with a BDD. This permits caching to be applied when f is computed, which reduces the time needed to compute a best transformer to approximately one day for each 8-bit operation. The approach does not scale to wider words nor to blocks.

Our work builds on that of Monniaux [20] who showed how transfer functions can be derived for operations over real-valued variables. His approach relies on universal quantifier elimination algorithm which is problematic for piecewise linear functions. Universal quantifier elimination also arises in work on inferring template constraints [12]. There the authors employ Farkas' lemma to transform universal quantification to existential quantification, albeit at the cost of compromising completeness (Farkas' lemma prevents integral reasoning).

The problem of handling limited precision arithmetic is discussed in [29]. Existing approaches are to: verify that no overflows arise using perfect numbers; revise the concretisation map to reflect truncation [29]; or deploy congruences [14,23] where the modulo is a power of two. Our work suggests handling wraps in the generation of the transfer functions, which we think is natural.

6 Concluding Discussion

This paper advocates deriving transfer functions from Boolean formulae since the elimination of universal quantifiers is trivial in this domain. Boolean formulae are natural candidates for expressing arithmetic, logical and bitwise operations, allowing transfer functions to be derived for blocks of code that would otherwise only be amenable to the coarsest of approximation. The paper shows how to distill transfer functions that are action systems of guarded updates, where the guards are octagonal inequalities and the updates are linear affine equalities. This formulation enables the application of a transfer function for a basic block to be reduced to a series of linear programming problems. Although we have illustrated the approach using octagons, there is no reason why richer classes of template constraints [28] could not be deployed to express the guards. Moreover, linear affine equalities could be substituted with polynomial equalities of degree at most d [22], say, and correspondingly linear programming replaced with non-linear programming. Thus the ideas presented in the paper generalise quite naturally. Finally, the approach will only become more attractive as linear solvers and SAT solvers continue to improve both in terms of efficiency and scalability.

Acknowledgements. We particularly thank David Monniaux for discussions at VMCAI 2010 in Madrid that motivated writing up the ideas in this paper. We thank Professor Stefan Kowalewski for his financial support that was necessary to initiate our collaboration. This work was also supported, in part, by a Royal Society industrial secondment and a Royal Society travel grant.

References

1. Atmel Corporation. The Atmel 8-bit AVR Microcontroller with 16K Bytes of In-system Programmable Flash (2009), http://www.atmel.com/atmel/acrobat/doc2466.pdf
2. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, USA (August 2007)
3. Chandru, V., Lassez, J.-L.: Qualitative Theorem Proving in Linear Constraints. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 395–406. Springer, Heidelberg (2004)
4. Chvátal, V.: Linear Programming. W. H. Freeman and Company, New York (1983)
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
6. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL, pp. 238–252. ACM Press, New York (1977)
7. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
8. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL, pp. 84–97. ACM Press, New York (1978)
9. Edmonds, J., Manrras, J.-F.: Note sur les Q-matrices d'Edmonds. Recherche Opérationnella 32(2), 203–209 (1997)
10. Granger, P.: Static Analysis of Arithmetical Congruences. International Journal of Computer Mathematics 30(13), 165–190 (1989)
11. Granger, P.: Static Analyses of Congruence Properties on Rational Numbers. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 278–292. Springer, Heidelberg (1997)
12. Gulwani, S., Srivastava, S., Venkatesan, R.: Program Analysis as Constraint Solving. In: PLDI, pp. 281–292. ACM Press, New York (2008)
13. Karr, M.: Affine Relationships among Variables of a Program. Acta Informatica 6, 133–151 (1976)
14. King, A., Søndergaard, H.: Automatic Abstraction for Congruences. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 197–213. Springer, Heidelberg (2010)
15. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
16. Le Berre, D.: SAT4J: Bringing the power of SAT technology to the Java platform (2010), http://www.sat4j.org/
17. Marriott, K.: Frameworks for Abstract Interpretation. Acta Informatica 30(2), 103–129 (1993)
18. Miné, A.: A New Numerical Abstract Domain Based on Difference-Bound Matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
19. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
20. Monniaux, D.: Automatic Modular Abstractions for Linear Constraints. In: POPL, pp. 140–151. ACM Press, New York (2009)
21. Monniaux, D.: Personal communication with Monniaux at VMCAI (January 2010)

22. Müller-Olm, M., Seidl, H.: A Note on Karr's Algorithm. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 1016–1028. Springer, Heidelberg (2004)
23. Müller-Olm, M., Seidl, H.: Analysis of Modular Arithmetic. ACM Trans. Program. Lang. Syst. 29(5) (August 2007)
24. Neumaier, A., Shcherbina, O.: Safe Bounds in Linear and Mixed-Integer Linear Programming. Math. Program. 99(2), 283–296 (2004)
25. Plaisted, D.A., Greenbaum, S.: A Structure-Preserving Clause Form Translation. Journal of Symbolic Computation 2(3), 293–304 (1986)
26. Regehr, J., Reid, A.: HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. ACM SIGOPS Operating Systems Review 38(5), 133–143 (2004)
27. Reps, T., Sagiv, M., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
28. Sankaranarayanan, S., Sipma, H., Manna, Z.: Constraint based linear relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)
29. Simon, A., King, A.: Taming the Wrapping of Integer Arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 121–136. Springer, Heidelberg (2007)

Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables

Alexandre Chapoutot

LIP6 - Université Pierre et Marie Curie
4, place Jussieu F-75252 Paris Cedex 05 France
alexandre.chapoutot@lip6.fr

Abstract. The design of embedded control systems is mainly done with model-based tools such as Matlab/Simulink. Numerical simulation is the central technique of development and verification of such tools. Floating-point arithmetic, which is well-known to only provide approximated results, is omnipresent in this activity. In order to validate the behaviors of numerical simulations using abstract interpretation-based static analysis, we present, theoretically and with experiments, a new partially relational abstract domain dedicated to floating-point variables. It comes from interval expansion of non-linear functions using slopes and it is able to mimic all the behaviors of the floating-point arithmetic. Hence it is adapted to prove the absence of run-time errors or to analyze the numerical precision of embedded control systems.

1 Introduction

Embedded control systems are made of a software and a physical environment which aim at continuously interact with each other. The design of such systems is usually realized with the model-based paradigm. Matlab/Simulink[1] is one of the most used tools for this purpose. It offers a convenient way to describe the software and the physical environment in an unified formalism. In order to verify that the control law, implemented in the software, fits the specification of the system, several numerical simulations are made under Matlab/Simulink. Nevertheless, this method is closer to test-based method than formal proof. Moreover, this verification method is strongly related to the floating-point arithmetic which provides approximated results.

Our goal is the use of abstract interpretation-based static analysis [9] to validate the design of control embedded software described in Matlab/Simulink. In our previous work [3], we defined an analysis to validate that the behaviors given by numerical simulations are close to the exact mathematical behaviors. It was based on an interval abstraction of floating-point numbers which may produce too coarse results. In this article, our work is focused on a tight representation of the behaviors of the floating-point arithmetic in order to increase the precision of the analysis of Matlab/Simulink models.

[1] Trademarks of The Mathworks™company.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 184–200, 2010.
© Springer-Verlag Berlin Heidelberg 2010

To emphasize the poor mathematical properties of the floating-point arithmetic, let us consider the sum of numbers given in Example 1 with a single precision floating-point arithmetic. The result of this sum is $-2.08616257.10^{-6}$ due to rounding errors, whereas the exact mathematical result is zero.

Example 1

$$0.0007 + (-0.0097) + 0.0738 + (-0.3122) + 0.7102 + (-0.5709) + (-1.0953) \\ + 3.3002 + (-2.9619) + (-0.2353) + 2.4214 + (-1.7331) + 0.4121$$

Example 1 shows that the summation of floating-point numbers is a very ill-conditioned problem [28, Chap. 6]. Indeed, small perturbations on the elements to sum produce a floating-point result which could be far from the exact result. Nevertheless, it is a very common operation in control embedded software. In particular, it is used in filtering algorithms or in regulation processes, such as for example in PID[2] regulation. Remark that depending on the case, the rounding errors may stay insignificant and the behaviors of floating-point arithmetic may be safe. In consequence, a semantic model of this arithmetic could be used to prove the behaviors of embedded control software using floating-point numbers.

The definition of abstract numerical domains for floating-point numbers is usually based on rational or real numbers [13,24] to cope with the poor mathematical structure of the floating-point set. In consequence, these domains give an over-approximation of the floating-point behaviors. This is because they do not bring information about the kind of numerical instability appearing during computations. We underline that our goal is not interested in computing the rounding errors but the floating-point result. In others words, we want to compute the bounds of floating-point variables without considering the numerical quality of these bounds.

Our main contribution is the definition of a new numerical abstract domain, called Floating-Point Slopes (FPS), dedicated to the study of floating-point numbers. It is based on interval expansion of non-linear functions named *interval slopes* introduced by Krawczyk and Neumaier [20] and, as we will show in this article, it is a partially relational domain. The main difference is that, in Proposition 1, we adapt the interval slopes to deal with floating-point numbers. Moreover, we are able to tightly represent the behaviors of floating-point arithmetic with our domain. A few cases studies will show the practical use of our domain. Hence we can prove properties on programs taking into account the behaviors of the floating-point arithmetic such that the absence of run-time errors or, by combining it with other domains *e.g.* [4], the quality of numerical computations.

Content. In Section 2, we will present the main features of floating-point arithmetic and we will also introduce the interval expansions of functions. We will present our abstract domain FPS in Section 3 and the analysis of floating-point programs in Section 4 before describing experimental results in Section 5. In Section 6, we will reference the related work before concluding in Section 7.

[2] PID stands for proportional-integral-derivative. It is a generic method of feedback loop control widely used in industry.

2 Background

We recall the main features of the IEEE754-2008 standard of floating-point arithmetic in Section 2.1. Next in Section 2.2, we present some results from interval analysis, in particular the interval expansion of functions.

2.1 Floating-Point Arithmetic

We briefly present the floating-point arithmetic, more details are available in [28] and the references therein. The IEEE754-2008 standard [18] defines the floating-point arithmetic in base 2 which is used in almost every computer[3].

Floating-point numbers have the following form: $f = s.m.2^e$. The value s represents the *sign*, the value m is the *significand* represented with p bits and the value e is the *exponent* of the floating-point number f which belongs into the interval $[e_{\min}, e_{\max}]$ such that $e_{\max} = -e_{\min} + 1$. There are two kinds of numbers in this representation. *Normalize numbers* for which the significand implicitly starts with a 1 and *denormalized numbers* that implicitly starts with a 0. The later are used to gain accuracy around zero by slowly degrading the precision.

The standard defines different values of p and $e_{\min}$: $p = 24$ and $e_{\min} = -126$ for the single precision and $p = 53$ and $e_{\min} = -1022$ for the double precision. We call *normal range* the set of absolute real values in $[2^{e_{\min}}, (2 - 2^{1-p})2^{e_{\max}}]$ and the *subnormal range* the set of numbers in $[0, 2^{e_{\min}}]$.

The set of floating-point numbers (single or double precision) is represented by $\mathbb{F}$ which is closed under negation. A few special values represent special cases: the values $-\infty$ and $+\infty$ to represent the negative or the positive overflow; and the value *NaN*[4] represents invalid results such that $\sqrt{-1}$.

The standard defines round-off functions which convert exact real numbers into floating-point numbers. We are mainly concerned by the rounding to the nearest ties to even[5] (noted fl), the rounding towards $+\infty$ and rounding toward $-\infty$. The round-off functions follow the correct rounding property, *i.e.* the result of a floating-point operation is the same that the rounding of the exact mathematical result. Note that these functions are monotone. We are interested in this article by computing the range of floating-point variables rounded to the nearest which is the default mode of rounding in computers.

A property of the round-off function fl is given in Equation (1). It characterizes the *overflow*, *i.e.* the rounding result is greater than the biggest element of $\mathbb{F}$ and the case of the generation of 0. This definition only uses positive numbers, using the symmetry property of $\mathbb{F}$, we can easily deduce the definition for the negative

[3] It also defines this arithmetic in base 10 but it is not relevant for our purpose.

[4] *NaN* stands for *Not A Number*.

[5] The IEEE754-2008 standard introduces two rounding modes to the nearest with respect to the previous IEEE754-1985 and IEEE754-1987 standards. These two modes only differ when an exact result is in half-way of two floating-point numbers. In rounding-to-nearest-tie-to-even mode, the floating-point number whose the least significand bit is even is chosen. Note that this definition is used in all the other revisions of the IEEE754 standard, see [28, Chap. 3.4] for more details.

part. We denote by $\sigma = 2^{e_{\min}-p+1}$ the smallest positive subnormal number and the largest finite floating-point number by $\Sigma = (2 - 2^{1-p})2^{e_{\max}}$.

$$\forall x \in \mathbb{F}, x > 0, \quad \mathrm{fl}(x) = \begin{cases} +0 & \text{if } 0 < x \leq \sigma/2 \\ +\infty & \text{if } x \geq \Sigma \end{cases} \tag{1}$$

An *underflow* [28, Sect. 2.3] is detected when the rounding result is less than $2^{e_{\min}}$, *i.e.* the result is in the subnormal range.

The errors associated to a correct rounding is defined in Equation (2) and it is valid for all floating-point numbers x and y except $-\infty$ and $+\infty$ (see [28, Chap. 2, Sect. 2.2]). The operation $\diamond \in \{+, -, \times, \div\}$ but it is also valid for the square root. The *relative rounding error unit* is denoted by μ. In single precision, $\mu = 2^{-24}$ and $\sigma = 2^{-149}$ and in double precision, $\mu = 2^{-53}$ and $\sigma = 2^{-1074}$.

$$\texttt{fl}(x \diamond y) = (x \diamond y)(1 + \epsilon_1) + \epsilon_2 \qquad \text{with } |\epsilon_1| \leq \mu \text{ and } |\epsilon_2| \leq \frac{1}{2}\sigma \tag{2}$$

If $\texttt{fl}(x \diamond y)$ is in the normal range or if $\diamond \in \{+, -\}$ then ϵ_2 is equal to zero. If $\texttt{fl}(x \diamond y)$ is in the subnormal range then ϵ_1 is equal to zero.

Numerical instabilities in programs come from the rounding representation of values and they also came from two problems due to finite precision:

Absorption. If $|x| \leq \mu|y|$ then it happens that $\mathrm{fl}(x+y) = \mathrm{fl}(y)$. For example, in single precision, the result of $\mathrm{fl}(10^4 - 10^{-4})$ is $\mathrm{fl}(10^4)$. In numerical analysis, the solution avoid this phenomenon is to sort the sequence of numbers [17, Chap. 4]. This solution is not applicable when the numbers to add are given by a sensor measuring the physical environment.

Cancellation. It appears in the subtraction $\mathrm{fl}(x - y)$ if $(|x - y|) \leq \mu(|x| + |y|)$ then the relative errors can be arbitrary big. Indeed, the rounding errors take usually place in the least significant digits of floating-point numbers. These errors may become preponderant in the result of a subtraction when the most significant digits of two closed numbers cancelled each others. In numerical analysis, subtraction of numbers coming from long computations are avoided to limit this phenomena. We cannot apply this solution in embedded control systems where some results are used at different instants of time.

2.2 Interval Arithmetic

We introduce interval arithmetic and in particular, the interval expansion of functions which is an element of our abstract domain FPS.

Standard Interval Arithmetic. The *interval arithmetic* [27] has been defined to avoid the problem of approximated results coming from the floating-point arithmetic. It had also been used as the first numerical abstract domain in [9].

When dealing with floating-point intervals the bounds have to be rounded to outward as in [24, Sect. 3]. In Example 2, we give the result of the interval evaluation in single precision of a sum of floating-point numbers.

Example 2. Using the interval domain for floating-point arithmetic [24, Sect. 3] the result of the sum defined by $\sum_{i=1}^{10} 10^1 + \sum_{i=1}^{10} 10^2 + \sum_{i=1}^{10} 10^3 + \sum_{i=1}^{1000} 10^{-3}$ is $[11100, 11101.953]$. The exact result is 11101 while the floating-point result is 11100 due to an absorption phenomena. The floating-point result and the exact result are in the result interval but we cannot distinguish them any more.

A source of over-approximation is known in the interval arithmetic as the *dependency problem* which is also known in static analysis as the non-relational aspect. For example, if some variable has value $[a, b]$, then the result of $x - x$ is $[a - b, b - a]$ which is equal to zero only if $a = b$. This problem is addressed by considering interval expansions of functions.

Notations. We denote by x a real number and by $\boldsymbol{x}$ a vector of real numbers. Interval values are in capital letters X or denoted by $[a, b]$ where a is the lower bound and b is the upper bound of the interval. A vector of interval values will be denoted by $\mathbf{X}$. We denote by $[\mathsf{f}]$ the interval extension of a function f obtained by substitution of all the arithmetic operations with their equivalent in interval. The center of an interval $[a, b]$ is represented by $\mathsf{mid}([a, b]) = a + 0.5 \times (b - a)$.

Extended Interval Arithmetic. We are interested in the computation of the image of a vector of interval $\mathbf{X}$ by a non-linear function $\mathsf{f} : \mathbb{R}^n \to \mathbb{R}$ only composed by additions, subtractions, multiplications and divisions and square root. In order to reduce over-approximations in the interval arithmetic, some interval expansions have been developed. The first one is based on the *Mean-Value Theorem* and it is expressed as:

$$\mathsf{f}(\mathbf{X}) \subseteq \mathsf{f}(\boldsymbol{z}) + [\mathsf{f}'](\mathbf{X})(\mathbf{X} - \boldsymbol{z}) \quad \forall \boldsymbol{z} \in \mathbf{X} \ . \tag{3}$$

The first-order approximation of the range of a function f can be defined thanks to its first order derivative f' over $\mathbf{X}$. We can then approximate $\mathsf{f}(\mathbf{X})$ by a pair $(\mathsf{f}(\boldsymbol{z}), [\mathsf{f}'](\mathbf{X}))$ that are the value of f at point z and the interval extension of f' evaluated over $\mathbf{X}$.

A second interval expansion has been defined by Krawczyk and Neumaier [20] using the notion of slopes which reduced the approximation of the derivative form. It is defined by the relation:

$$\begin{aligned} &\mathsf{f}(\mathbf{X}) \subseteq \mathsf{f}(\boldsymbol{z}) + [\mathtt{F}^{\boldsymbol{z}}](\mathbf{X})(\mathbf{X} - \boldsymbol{z}) \\ &\qquad \text{with } \mathtt{F}^{\boldsymbol{z}}(\mathbf{X}) = \left\{ \frac{\mathsf{f}(\boldsymbol{x}) - \mathsf{f}(\boldsymbol{z})}{\boldsymbol{x} - \boldsymbol{z}} : \boldsymbol{x} \in \mathbf{X} \wedge \boldsymbol{z} \neq \boldsymbol{x} \right\} \ . \end{aligned} \tag{4}$$

Then we can represent $\mathsf{f}(\mathbf{X})$ by a pair $(\mathsf{f}(\boldsymbol{z}), [\mathtt{F}^{\boldsymbol{z}}](\mathbf{X}))$ that are the value of f in the point $\boldsymbol{z}$ and the interval extension of the slope $\mathtt{F}^{z}(X)$ of f.

Note that the value $\boldsymbol{z}$ is constructed, in general, from the centers of the interval variables appearing in the function f for both interval expansions.

An interesting feature is that we can inductively compute the derivative or the slope of a functions using *automatic differentiation* techniques [1]. It is a

semantic-based method to compute derivatives. In this context, we call *independent variables* some input variables of a program with respect to which derivatives are computed. We call *dependent variables* output variables whose derivatives are desired. A *derivative object* represents derivative information, such as a vector of partial derivatives like $(\partial e/\partial x_1, \dots, \partial e/\partial x_n)$ of some expression e with respect to a vector $\boldsymbol{x}$ of independent variables. The main idea of automatic differentiation is that every complicated function f, *i.e.* a program, is composed by simplest elements, *i.e.* program instructions. Knowing the derivatives of these elements with respect to some independent variables, we can compute the derivatives or the slopes of f following the differential calculus rules. Furthermore, using interval arithmetic in the differential calculus rules, we can guarantee the result.

We give in Table 1 the rules to compute derivatives or slopes with respect to the structure of arithmetic expressions. We assume that we know the number of independent variables in the programs and we denote by n this number. The variable $\mathcal{V}^{\text{ind}}$ represents the vector of independent variables with respect to which the derivatives are computed. We denote by δ_i the interval vector of length n, having all its coordinates equal to $[0, 0]$ except the i-th element equals to $[1, 1]$. So, we consider that all the independent variables are assigned to a unique position i in $\mathcal{V}^{\text{ind}}$ and it is initially assigned with a derivative object equal to δ_i. Following Table 1 where g and h represent variables with derivative object, a constant value c has a derivative object equal to zero (the interval vector $\mathbf{0}$ has all its coordinates equal to $[0, 0]$). For addition and subtraction, the result is the vector addition or the vector subtraction of the derivative objects. For multiplication and division, it is more complicated but the rules come from the standard rules of the composition of derivatives, *e.g.* $(u \times v)' = u' \times v + u \times v'$. A proof of the computation rules[6] for slopes can be found in [30, Sect. 1]. Note that we can apply automatic differentiation for other functions, such as the square root, using the rule of function composition, $(f \circ g)'(x) = f'(g(x))g'(x)$.

These interval expansions of functions, using either $(\mathsf{f}(\boldsymbol{z}), [\mathsf{f}'](\mathbf{X}))$ the derivative form or $(\mathsf{f}(\boldsymbol{z}), [\mathtt{f}^{\boldsymbol{z}}](\mathbf{X}))$ the slope form, define a straightforward semantics of arithmetic expressions which can be used to compute bounds of variables.

Table 1. Automatic differentiation rules for derivatives and slopes

Function	Derivative arithmetic	Slope arithmetic
$c \in \mathbb{R}$	$\mathbf{0}$	$\mathbf{0}$
$\mathsf{g} + \mathsf{h}$	$[\mathsf{g}'](\mathbf{X}) + [\mathsf{h}'](\mathbf{X})$	$[\mathtt{G}^{\boldsymbol{z}}](\mathbf{X}) + [\mathtt{H}^{\boldsymbol{z}}](\mathbf{X})$
$\mathsf{g} - \mathsf{h}$	$[\mathsf{g}'](\mathbf{X}) - [\mathsf{h}'](\mathbf{X})$	$[\mathtt{G}^{\boldsymbol{z}}](\mathbf{X}) - [\mathtt{H}^{\boldsymbol{z}}](\mathbf{X})$
$\mathsf{g} \times \mathsf{h}$	$[\mathsf{g}'](\mathbf{X}) \times \mathsf{h}(\mathbf{X}) + \mathsf{g}(\mathbf{X}) \times [\mathsf{h}'](\mathbf{X})$	$[\mathtt{G}^{\boldsymbol{z}}](\mathbf{X}) \times \mathsf{h}(\mathbf{X}) + \mathsf{g}(\boldsymbol{z}) \times [\mathtt{H}^{\boldsymbol{z}}](\mathbf{X})$
$\dfrac{\mathsf{g}}{\mathsf{h}}$	$\dfrac{[\mathsf{g}'](\mathbf{X}) \times \mathsf{h}(\mathbf{X}) - [\mathsf{h}'](\mathbf{X}) \times \mathsf{g}(\mathbf{X})}{\mathsf{h}^2(\mathbf{X})}$	$\dfrac{[\mathtt{G}^{\boldsymbol{z}}](\mathbf{X}) - [\mathtt{H}^{\boldsymbol{z}}](\mathbf{X}) \times \frac{\mathtt{g}(\boldsymbol{z})}{\mathtt{h}(\boldsymbol{z})}}{\mathsf{h}(\mathbf{X})}$
$\sqrt{\mathsf{g}}$	$\dfrac{1}{2}\dfrac{[\mathsf{g}'](\mathbf{X})}{\sqrt{\mathsf{g}(\mathbf{X})}}$	$\dfrac{[\mathtt{G}^{\boldsymbol{z}}](\mathbf{X})}{\sqrt{g(\boldsymbol{z})} + \sqrt{g(\mathbf{X})}}$

[6] In [20, Sect. 2], the authors went also into detail of the complexity of these operations.

Remark 1. The difference in over-approximated result between the derivative form and the slope form is in the multiplication and the division rules. In the derivative form, we need to evaluate the two operands (g and h) using interval arithmetic while we only need to evaluate one of them in the slope form. Note also that we could have defined the multiplication by $[\mathtt{H}^{\boldsymbol{z}}](\mathbf{X}) \times \mathsf{g}(\mathbf{X}) + \mathsf{h}(\boldsymbol{z}) \times [\mathtt{G}^{\boldsymbol{z}}](\mathbf{X})$ (the division has also two forms) but the two possible forms of slope are over-approximations of $\mathsf{f}(\boldsymbol{X})$. Nevertheless, a possible way to choose between the two forms is to keep the form which gives the smallest approximation of $\mathsf{f}(\boldsymbol{X})$.

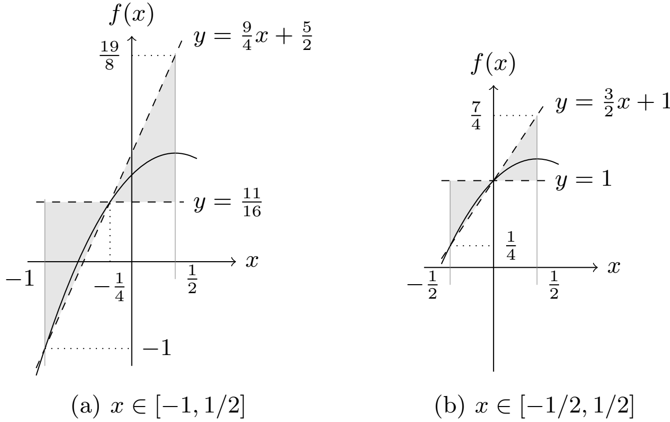


Fig. 1. Two examples of the interval expansion with slopes

In Figure 1, we give two graphical representations of interval slope expansion. For this purpose, we want to compute the image of x by the function $f(x) = x(1-x)+1$. We consider in Figure 1(a) that $x \in [-1, 1/2]$ and we get as a result that $f(x) \in [-1, 19/8]$ which is an over-approximation of the exact result $[-1, 5/4]$. The midpoint is $-1/4$ and the set of slopes is bounded by the interval $[0, 9/4]$. The dashed lines represent the linear approximation of the image. In Figure 1(b), we consider that $x \in [-1/2, 1/2]$ and the result is $f(x) \in [1/4, 7/4]$ which is still an over-approximation of the exact result $[1/4, 5/4]$. In that case, the midpoint is 0 and the set of slopes is bounded by the interval $[0, 3/2]$. Note that the smaller the interval the better the approximation is.

Example 3 shows that we can encode with interval slopes the list of variables contributing in the result of an arithmetic expression. In particular, the vector composing the interval slope of the variable t represents the influence of the variables a, b and c on the value of t. For example, we know that a modification of the value of the variable a produce a modification of the result with the same order of the modification on a because the slope associated to a is $[1, 1]$. But a modification on the variable b by $\varDelta_b$ will produce a modification on the t by $\varDelta_b \times \mathrm{V}_c$ because the slope of b is equal to V_c.

Example 3. Let $t = a + b \times c$, we want to compute the interval slope $[\mathtt{T}^z](\mathbf{X})$ of t. We consider that $\mathcal{V}^{\text{ind}} = \{a, b, c\}$ and $\mathbf{X}$ is the interval vector of the values of these variables. We suppose that the interval slope expansion of a, b and c are $(z_a, [\mathtt{A}^{\boldsymbol{z}}](\mathbf{X}) = \delta_1)$, $(z_b, [\mathtt{B}^{\boldsymbol{z}}](\mathbf{X}) = \delta_2)$, and $(z_c, [\mathtt{C}^{\boldsymbol{z}}](\mathbf{X}) = \delta_3)$ respectively. The interval value associated to c is V_c *i.e.* $\mathrm{V}_c = z_c + [\mathtt{C}^z](\mathbf{X})(\mathbf{X} - \boldsymbol{z})$.

$$\begin{aligned}
[\mathtt{T}^{\boldsymbol{z}}](\mathbf{X}) &= [\mathtt{A}^{\boldsymbol{z}}](\mathbf{X}) + z_b[\mathtt{C}^{\boldsymbol{z}}](\mathbf{X}) + [\mathtt{B}^{\boldsymbol{z}}](\mathbf{X})\,(z_c + [\mathtt{C}^{\boldsymbol{z}}](\mathbf{X})(\mathbf{X} - \boldsymbol{z})) \\
&= ([1,1], 0, 0) + z_b \times (0, 0, [1,1]) + (0, [1,1], 0) \times \mathrm{V}_c \\
&= ([1,1], [1,1] \times \mathrm{V}_c, z_b \times [1,1]) \\
&= ([1,1], \mathrm{V}_c, [z_b, z_b])
\end{aligned}$$

As seen in Example 3, interval slopes represent relations between the inputs and the outputs of a function. By computing interval slopes, we build step by step the set of variables related to arithmetic expressions in programs. In static analysis, we can use this interval expansion to track the influence of the inputs of a program on its outputs. Hence the choice of the set $\mathcal{V}^{\text{ind}}$ of independent variables is given by the set of the input variables of the program to analyse. Moreover, we can add in $\mathcal{V}^{\text{ind}}$ all the other variables which may influence output.

3 Floating-Point Slopes

We present in this section our new abstract domain FPS. In Section 3.1, we adapt the computation rules of interval slopes to take into account floating-point arithmetic. Next in Section 3.2, we define an abstract semantics of arithmetic expressions over FPS values taking into account the behaviors of floating-point arithmetic. And in Section 3.3, we define the order structure of the FPS domain.

3.1 Floating-Point Version of Interval Slopes

The definition of interval slope expansion in Section 2.2 manipulates real numbers. In case of floating-point numbers, we have to take into account the round-off function and the rounding-errors.

We show in Proposition 1 that the range of a non-linear function f of floating-point numbers can be soundly over-approximated by a floating-point slope. The function f must respect the correct rounding, *i.e.* the property of Equation (2) must hold. In other words, the result of an operation over set of floating-point numbers is over-approximated by the result of the same operation over floating-point slopes by adding a small quantity depending on the relative rounding error unit μ and the absolute error σ.

Proposition 1. *Let* $\mathsf{f} : D \subseteq \mathbb{R}^n \to \mathbb{R}$ *be an arithmetic operation of the form* $g \diamond h$ *with* $\diamond \in \{+, -, \times, \div\}$ *or* $\sqrt{\ }$, i.e. f *respects the correct rounding. For all* $\mathbf{X} \subseteq D$ *and* $\boldsymbol{z} \in D$*, we have:*

$$\mathrm{fl}\big(\mathsf{f}(\mathbf{X})\big) \subseteq \mathsf{f}(\boldsymbol{z})\big(1 + [-\mu, \mu]\big) + \left[-\frac{\sigma}{2}, \frac{\sigma}{2}\right] + [\mathtt{F}^{\boldsymbol{z}}](\mathbf{X})(X - z)\big(1 + [-\mu, \mu]\big) \ .$$

Proof

$$\begin{aligned}
\mathrm{fl}\big(\mathsf{f}(\mathbf{X})\big) &= \{\mathsf{f}(\boldsymbol{x})(1+\varepsilon_x)+\bar{\varepsilon}_x : \boldsymbol{x}\in\mathbf{X}\} && \textit{by Eq. (2)}\\
&\subseteq \mathsf{f}(\mathbf{X})+\mathsf{f}(\mathbf{X})\{\varepsilon_x : \boldsymbol{x}\in\mathbf{X}\}+\{\bar{\varepsilon}_x : \boldsymbol{x}\in\mathbf{X}\}\\
&\subseteq \big(\mathsf{f}(\boldsymbol{z})+[\mathtt{F}^{\boldsymbol{z}}](\mathbf{X})(\mathbf{X}-\boldsymbol{z})\big)+\{\bar{\varepsilon}_x : \boldsymbol{x}\in\mathbf{X}\} && \textit{by Eq. (4)}\\
&\quad+\big(\mathsf{f}(\boldsymbol{z})+[\mathtt{F}^{\boldsymbol{z}}](\mathbf{X})(\mathbf{X}-\boldsymbol{z})\big)\{\varepsilon_x : \boldsymbol{x}\in\mathbf{X}\}\\
&\subseteq \mathsf{f}(\boldsymbol{z})\big(1+\{\varepsilon_x : \boldsymbol{x}\in\mathbf{X}\}\big)+\{\bar{\varepsilon}_x : \boldsymbol{x}\in\mathbf{X}\}\\
&\quad+[\mathtt{F}^{\boldsymbol{z}}](\mathbf{X})(\mathbf{X}-\boldsymbol{z})\big(1+\{\varepsilon_x : \boldsymbol{x}\in\mathbf{X}\}\big)\\
&\subseteq \mathsf{f}(\boldsymbol{z})\big(1+[-\mu,\mu]\big)+\Big[-\frac{\sigma}{2},\frac{\sigma}{2}\Big] && |\varepsilon_x|\le\mu \textit{ by Eq. (2)}\\
&\quad+[\mathtt{F}^{\boldsymbol{z}}](\mathbf{X})(\mathbf{X}-\boldsymbol{z})\big(1+[-\mu,\mu]\big) && |\bar{\varepsilon}_x|\le\frac{1}{2}\sigma \textit{ by Eq. (2)}
\end{aligned}$$

□

Remark 2. As the floating-point version of slopes is based on μ and σ, we can represent the floating-point behaviors depending of the hardware. For example, extended precision[7] is represented using the values $\mu = 2^{-64}$ and $\sigma = 2^{-16446}$. Furthermore following [2], we can compute the result of a double rounding[8] with $\mu = (2^{11}+2)2^{-64}$ and $\sigma = (2^{11}+1)2^{-1086}$.

Proposition 1 shows that we can compute the floating-point range of a function f, respecting the correct rounding, using interval slopes expansion. That is a set of floating-point values can is represented by a pair:

$$\Big([\mathsf{f}]\,(\mathbf{z})\big(1+[-\mu,\mu]\,\big)+\Big[-\frac{\sigma}{2},\frac{\sigma}{2}\Big],\quad [\mathtt{F}^{\boldsymbol{z}}](\boldsymbol{X})\big(1+[-\mu,\mu]\big)\Big)\ .$$

The first element is a small interval rounding to the nearest around $\mathsf{f}(\boldsymbol{z})$ for which we have to take into account the possible rounding errors. The second element is the interval slopes which have to take account of relative errors. Note that this adaptation adds a very little overhead of computations compared to the definition of interval slopes by Krawczyk and Neumaier.

3.2 Semantics of Arithmetic Operations

In this section, we define the abstract semantics of arithmetic operations over elements of floating-point slopes domain in order to mimic the behaviors of the floating-point arithmetic. We denote by $\mathbb{I}$ the set of intervals and by $\mathbb{S} = \mathbb{I}\times\mathbb{I}^{|\mathcal{V}^{\mathrm{ind}}|}$ the set of slopes. An element s of $\mathbb{S}$ is represented by a pair $(\mathrm{M},\mathbf{S})$ where M is a floating-point interval and $\mathbf{S}$ is a vector of floating-point intervals. We denote by $\langle\mathbb{I},\sqsubseteq_{\mathbb{I}},\bot_{\mathbb{I}},\top_{\mathbb{I}},\sqcup_{\mathbb{I}},\sqcap_{\mathbb{I}}\rangle$ the lattice of intervals. First we define some auxiliary functions before presenting the semantics of arithmetic expressions over FPS.

---

[7] In some hardware, *e.g.* Intel x87, floating-point numbers may be encoded with 80 bits in registers, *i.e.* the significand is 64 bits long.

[8] It may happen on hardware using extended precision. Results of computations are rounded in registers and they are rounded again, with a less precision, in memory.

The function ι defined in Equation (5) computes the interval value associated to a floating-point slopes $(\mathrm{M}, \mathbf{S})$. We assume that the values of independent variables are kept in a separate interval vector $\mathbf{V}_{\mathcal{V}^{\text{ind}}}$. The notation $\mathsf{mid}(\mathbf{V}_{\mathcal{V}^{\text{ind}}})$ stands for the component-wise application of the function mid on all the components of the vector $\mathbf{V}_{\mathcal{V}^{\text{ind}}}$. Note that $\cdot$ represents the scalar product.

$$\iota\big((\mathrm{M}, \mathbf{S})\big) = \mathrm{M} + \mathbf{S} \cdot \big(\mathbf{V}_{\mathcal{V}^{\text{ind}}} - \mathsf{mid}(\mathbf{V}_{\mathcal{V}^{\text{ind}}})\big) \tag{5}$$

The function κ defined in Equation (6) transforms an interval value $[a, b]^{\ell}$ associated to the ℓ-th independent variable into a floating-point slope.

$$\kappa\left([a, b]^{\ell}\right) = \big([m, m], \delta_{\ell}\big) \quad \text{with} \quad m = \mathsf{mid}([a, b]) \tag{6}$$

This function κ is used in two cases: *i)* To initialize all the independent variables at the beginning of an analysis. *ii)* In the meet operation, see Section 3.3.

We can detect overflows and generations of zero by using the function Φ defined in Equation (7). We have two kinds or rules: *total* rules when we are certain that a zero or an overflow occur and *partial* rules when a part of the set described by a floating-point slope generates a zero or an overflow. With the function ι we can determine for an element $(\mathrm{M}, \mathbf{S}) \in \mathbb{S}$ if $(\mathrm{M}, \mathbf{S})$ represents an overflow or a zero. Hence we represent the finite precision of the floating-point arithmetic. We denote by $\mathbf{p}_{\infty}$ and by $\mathbf{m}_{\infty}$ the interval vectors with all their components equal to $[+\infty, +\infty]$ and $[-\infty, -\infty]$ respectively. We recall that σ is the smallest denormalized and Σ is the largest floating-point numbers.

$$\Phi(\mathrm{M}, \mathbf{S}) = \begin{cases} ([0,0], \mathbf{0}) & \text{if } \iota(\mathrm{M}, \mathbf{S}) \sqsubseteq_{\mathbb{I}} [-\frac{\sigma}{2}, \frac{\sigma}{2}] \\ (\tilde{\mathrm{M}}, \mathbf{0} \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \iota(\mathrm{M}, \mathbf{S}) \sqcap_{\mathbb{I}} \,]-\frac{\sigma}{2}, \frac{\sigma}{2}[\, \neq \bot_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}} = \begin{cases} [0,0] & \text{if } \mathrm{M} \sqsubseteq_{\mathbb{I}} \,]-\frac{\sigma}{2}, \frac{\sigma}{2}[\\ [0,0] \sqcup_{\mathbb{I}} \mathrm{M} & \text{otherwise} \end{cases} \\ ([+\infty, +\infty], \mathbf{p}_{\infty}) & \text{if } \iota(\mathrm{M}, \mathbf{S}) \sqsubseteq_{\mathbb{I}} \,]\Sigma, +\infty] \\ (\tilde{\mathrm{M}}, \mathbf{p}_{\infty} \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \iota(\mathrm{M}, \mathbf{S}) \sqcap_{\mathbb{I}} \,]\Sigma, +\infty] \neq \bot_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}} = \begin{cases} [+\infty, +\infty] & \text{if } \mathrm{M} \sqsubseteq_{\mathbb{I}} \,]\Sigma, +\infty] \\ [+\infty, +\infty] \sqcup_{\mathbb{I}} \mathrm{M} & \text{otherwise} \end{cases} \\ ([-\infty, -\infty], \mathbf{m}_{\infty}) & \text{if } \iota(\mathrm{M}, \mathbf{S}) \sqsubseteq_{\mathbb{I}} [-\infty, -\Sigma[\\ (\tilde{\mathrm{M}}, \mathbf{m}_{\infty} \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \iota(\mathrm{M}, \mathbf{S}) \sqcap_{\mathbb{I}} [-\infty, -\Sigma[\, \neq \bot_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}} = \begin{cases} [-\infty, -\infty] & \text{if } \mathrm{M} \sqsubseteq_{\mathbb{I}} [-\infty, -\Sigma[\\ [-\infty, -\infty] \sqcup_{\mathbb{I}} \mathrm{M} & \text{otherwise} \end{cases} \\ (\mathrm{M}, \mathbf{S}) & \text{otherwise} \end{cases} \tag{7}$$

Equation (7) is an adaptation of the rule defined in Equation (1) to deal with FPS values. Furthermore, the abstract values $(+\infty, \mathbf{p}_{\infty})$ and $(-\infty, \mathbf{m}_{\infty})$ represent the special floating-point values $+\infty$ and $-\infty$ respectively. As in floating-point arithmetic, the values $(+\infty, \mathbf{p}_{\infty})$ and $(-\infty, \mathbf{m}_{\infty})$ are absorbing elements.

An interesting feature of interval slopes is that we can mimic the absorption phenomenon by setting to zero the interval slope of the absorbed operand. We define the function ρ for this purpose. Indeed, an abstract value $(\mathrm{M}, \mathbf{S})$ already supports partial absorption as M is computed with a rounding to the nearest but

$\mathbf{S}$ have to be *reduced* to represent the absence of the influence of particular independent variables. The reduction of an abstract value $g = (\mathrm{M}_g, \mathbf{S}_g)$ compared to an abstract value $h = (\mathrm{M}_h, \mathbf{S}_h)$, denoted by $\rho(g \mid h)$, is defined in Equation (8).

$$\rho(g \mid h) = \begin{cases} ([0,0], \mathbf{0}) & \text{if } \iota\big(\mathrm{M}_g, \mathbf{S}_g\big) \sqsubseteq_{\mathbb{I}} [\mu, \mu] \times \iota\big(\mathrm{M}_h, \mathbf{S}_h\big) \\ \big(\tilde{\mathrm{M}}_g, \mathbf{0} \,\dot{\sqcup}_{\mathbb{I}}\, \mathbf{S}_g\big) & \text{if } \iota\big(\mathrm{M}_g, \mathbf{S}_g\big) \sqcap_{\mathbb{I}} [\mu, \mu] \times \iota\big(\mathrm{M}_h, \mathbf{S}_h\big) \neq \bot_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}}_g = \begin{cases} [0,0] & \text{if } \mathrm{M}_g \sqsubseteq_{\mathbb{I}} [\mu, \mu] \times \iota\big(\mathrm{M}_h, \mathbf{S}_h\big) \\ [0,0] \sqcup_{\mathbb{I}} \mathrm{M}_g & \text{otherwise} \end{cases} \\ \big(\mathrm{M}_g, \mathbf{S}_g\big) & \text{otherwise} \end{cases} \tag{8}$$

Equation (8) models the absorption phenomenon by explicitly setting to zero the values of a slope. As mentioned in Section 2.2, a slope shows which variables influence the computation of an arithmetic expression. But, absorption phenomena induce that an operand does not influence the result of an addition or a subtraction any more.

Using the functions Φ, ρ and ι, we inductively define on the structure of arithmetic expressions the abstract semantics $[\![.]\!]^\sharp_{\mathbb{S}}$ of floating-point slopes in Figure 2. We denote by $\mathrm{env}^\sharp$ an abstract environment which associates to each program variable a floating-point slope. For each arithmetic operation, we component-wisely combine the elements of the abstract operands $[\![g]\!]^\sharp_{\mathbb{S}}(\mathrm{env}^\sharp) = (\mathrm{M}_g, \mathbf{S}_g)$ and $[\![h]\!]^\sharp_{\mathbb{S}}(\mathrm{env}^\sharp) = (\mathrm{M}_h, \mathbf{S}_h)$. The element M is obtained using the interval arithmetic with rounding to the nearest. The element $\mathbf{S}$ is computed using the definition of the slope arithmetic defined in Table 1. We take into account of the possible rounding errors in the result $(\mathrm{M}, \mathbf{S})$ following Proposition 1. In case of addition and subtraction, according to the Equation (2), we do not consider absolute error $\frac{\sigma}{2}$ which is always zero. Moreover, in case of addition or subtraction, we handle the absorption phenomena using the function ρ, defined in Equation (8). Finally, we check if a zero or an overflow is generated by applying the function Φ defined in Equation (7).

Remark 3. The functions Φ and ρ make the arithmetic operations on floating-point slopes non associative and non distributive as in floating-point arithmetic.

3.3 Order Structure

In this section, we define the order structure of the set $\mathbb{S}$ of floating-point slopes. In particular, this structure is based on the lattice of intervals. We recall that the set of slopes $\mathbb{S} = \mathbb{I} \times \mathbb{I}^{|\mathcal{V}^{\mathrm{ind}}|}$ and an element s of $\mathbb{S}$ is a pair $(\mathrm{M}, \mathbf{S})$.

We define a partial order, the join and the meet operations between elements of $\mathbb{S}$. All these operations are defined as a component-wise application of the associated operations of the interval domain except the meet operation which needs extra care. We denote by $\dot{\sqsubseteq}_{\mathbb{I}}$ the component-wise application of the interval order. We can define a partial order $\sqsubseteq_{\mathbb{S}}$ between elements of $\mathbb{S}$ with:

$$\forall (\mathrm{M}_g, \mathbf{S}_g), (\mathrm{M}_h, \mathbf{S}_h) \in \mathbb{S}, (\mathrm{M}_g, \mathbf{S}_g) \sqsubseteq_{\mathbb{S}} (\mathrm{M}_h, \mathbf{S}_h) \Leftrightarrow \mathrm{M}_g \sqsubseteq_{\mathbb{I}} \mathrm{M}_h \wedge \mathbf{S}_g \;\dot{\sqsubseteq}_{\mathbb{I}}\; \mathbf{S}_h \ . \tag{9}$$

$$\llbracket g \pm h \rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left((\tilde{\mathrm{M}}_g \pm \tilde{\mathrm{M}}_h)(1+[-\mu,\mu]),\quad \left(\tilde{\mathbf{S}}_g \pm \tilde{\mathbf{S}}_h\right)(1+[-\mu,\mu])\right)$$
$$\text{with}\quad (\tilde{\mathrm{M}}_g, \tilde{\mathbf{S}}_g) = \rho(g \mid h) \text{ and } (\tilde{\mathrm{M}}_h, \tilde{\mathbf{S}}_h) = \rho(h \mid g)$$
$$\llbracket g \times h \rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left(\mathrm{M},\quad \left(\mathbf{S}_g \times \iota(\mathrm{M}_h, \mathbf{S}_h) + \mathrm{M}_g \times \mathbf{S}_h\right)(1+[-\mu,\mu])\right)$$
$$\text{with}\quad \mathrm{M} = (\mathrm{M}_g \times \mathrm{M}_h)(1+[-\mu,\mu]) + \left[\frac{\sigma}{2}, \frac{\sigma}{2}\right]$$
$$\left\llbracket \frac{g}{h} \right\rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left(\mathrm{M},\quad \frac{\mathbf{S}_g - \mathbf{S}_h \frac{\mathrm{M}_g}{\mathrm{M}_h}}{\iota(\mathrm{M}_h, \mathbf{S}_h)}(1+[-\mu,\mu])\right)$$
$$\text{with}\quad \mathrm{M} = \frac{\mathrm{M}_g}{\mathrm{M}_h}(1+[-\mu,\mu]) + \left[\frac{\sigma}{2}, \frac{\sigma}{2}\right],$$
$$0 \notin \iota(\mathrm{M}_h, \mathbf{S}_h) \text{ and } 0 \notin \mathrm{M}_h$$
$$\llbracket \sqrt{g} \rrbracket^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left(\mathrm{M},\quad \left(\frac{\mathbf{S}_g}{\sqrt{\mathrm{M}_g} + \sqrt{\iota(\mathrm{M}_g, \mathbf{S}_g)}}\right)(1+[-\mu,\mu])\right)$$
$$\text{with}\quad \mathrm{M} = \left(\sqrt{\mathrm{M}_g}\,(1+[-\mu,\mu])\right) + \left[-\frac{\sigma}{2}, \frac{\sigma}{2}\right],$$
$$\mathrm{M}_g \sqcap_{\mathbb{I}} [-\infty, 0] = \bot_{\mathbb{I}} \text{ and } \iota(\mathrm{M}_g, \mathbf{S}_g) \sqcap_{\mathbb{I}} [-\infty, 0] = \bot_{\mathbb{I}}$$

Fig. 2. Abstract semantics of arithmetic expressions on floating-point slopes

The join operation $\sqcup_{\mathbb{S}}$ over floating-point slopes is defined in Equation (10). We denote by $\dot{\sqcup}_{\mathbb{I}}$ the component-wise application of the operation $\sqcup_{\mathbb{I}}$.

$$\forall (\mathrm{M}_g, \mathbf{S}_g), (\mathrm{M}_h, \mathbf{S}_h) \in \mathbb{S},\quad \left(\mathrm{M}_g, \mathbf{S}_g\right) \sqcup_{\mathbb{S}} \left(\mathrm{M}_h, \mathbf{S}_h\right) = \left(\mathrm{M}, S\right)$$
$$\text{with}\quad \mathrm{M} = \mathrm{M}_g \sqcup_{\mathbb{I}} \mathrm{M}_h \quad\text{and}\quad S = \mathbf{S}_g \,\dot{\sqcup}_{\mathbb{I}}\, \mathbf{S}_h \tag{10}$$

There is no direct way to define the greatest lower bound of two elements of $\mathbb{S}$. Indeed, two abstract values may represent the same concrete value but without being comparable. Hence we only have a join-semilattice structure. The meet operation $\sqcap_{\mathbb{S}}$ over floating-point slopes is defined in Equation (11). It may require a conversion into interval value. We consider that the result of the meet operation introduces a new independent variable at index ℓ. We denote by $\sqsubset_{\mathbb{I}}$ the strict comparison of intervals and by $\bot_{\mathbb{S}}$ the least element of $\mathbb{S}$.

$$\forall (\mathrm{M}_g, \mathbf{S}_g), (\mathrm{M}_h, \mathbf{S}_h) \in \mathbb{S},\quad \left(\mathrm{M}_g, \mathbf{S}_g\right) \sqcap_{\mathbb{S}} \left(\mathrm{M}_h, \mathbf{S}_h\right) =$$
$$\begin{cases} \bot_{\mathbb{S}} & \text{if } \iota(\mathrm{M}_g, \mathbf{S}_g) \sqcap_{\mathbb{I}} \iota(\mathrm{M}_h, \mathbf{S}_h) = \bot_{\mathbb{I}} \\ (\mathrm{M}_g, \mathbf{S}_g) & \text{if } \iota(\mathrm{M}_g, \mathbf{S}_g) \sqsubset_{\mathbb{I}} \iota(\mathrm{M}_h, \mathbf{S}_h) \\ (\mathrm{M}_h, \mathbf{S}_h) & \text{if } \iota(\mathrm{M}_h, \mathbf{S}_h) \sqsubset_{\mathbb{I}} \iota(\mathrm{M}_g, \mathbf{S}_g) \\ \kappa\big(\iota(\mathrm{M}_h, \mathbf{S}_h) \sqcap^{\ell}_{\mathbb{I}} \iota(\mathrm{M}_g, \mathbf{S}_g)\big) & \text{otherwise} \end{cases} \tag{11}$$

Note on the Widening Operator. In order to enforce the convergence of the fixpoint computation, we can define a widening operation $\nabla_{\mathbb{S}}$ over floating-point

slopes values. An advantage of our domain is that we can straightforwardly use the widening operations defined for the interval domain denoted by $\nabla_{\mathbb{I}}$. We define the operator $\nabla_{\mathbb{S}}$ in Equation (12) using the widening operator between intervals. The notation $\dot{\nabla}_{\mathbb{I}}$ represents the component-wise application of $\nabla_{\mathbb{I}}$ between the components of the interval slopes vector.

$$\forall (\mathrm{M}_g, \mathbf{S}_g), (\mathrm{M}_h, \mathbf{S}_h) \in \mathbb{S}, \quad \big(\mathrm{M}_g, \mathbf{S}_g\big) \nabla_{\mathbb{S}} \big(\mathrm{M}_h, \mathbf{S}_h\big) = \big(\mathrm{M}, \mathbf{S}\big) \\ \text{with} \quad \mathrm{M} = \mathrm{M}_g \; \nabla_{\mathbb{I}} \; \mathrm{M}_h \quad \text{and} \quad \mathbf{S} = \mathbf{S}_g \; \dot{\nabla}_{\mathbb{I}} \; \mathbf{S}_h \tag{12}$$

4 Analysis of Floating-Point Programs

The goal of the static analysis of floating-point programs using the floating-point slopes domain is to give for each control point and for each variable an over-approximation given by FPS of the reachable set of floating-point numbers. An abstract environment $\text{env}^\sharp$ associates to each variable $v \in \mathcal{V}$ a value of $\mathbb{S}$. The set $\mathcal{V}$ is made of the sets $\mathcal{V}^{\text{ind}}$ and $\mathcal{V}^{\text{dep}}$ of independent and dependent variables.

The semantics of an assignment $[\![v := e]\!]^\sharp$ in the abstract environment $\text{env}^\sharp$ is the update of the value associated to v with the result of the evaluation of the arithmetic expression e using the arithmetic operations over FPS given in Figure 2. As the FPS domain is related to the interval domain we can straightforwardly use the semantics of tests given in [15] to refine the value of variables. Note that the semantics of tests is related to the meet operation defined in Equation (11) which may conserve some relations between variables.

We define in Equation (13) the concretization function $\gamma_{\mathbb{S}}$ between the join-semilattice $\langle \mathcal{V} \to \mathbb{S}, \dot{\sqsubseteq}_{\mathbb{S}} \rangle$, with $\dot{\sqsubseteq}_{\mathbb{S}}$ the point-wise lifting comparison, and the complete lattice $\langle \wp(\mathcal{V} \to \mathbb{F}), \subseteq \rangle$.

$$\gamma_{\mathbb{S}}\big(v \mapsto (\mathrm{M}, \mathbf{S})\big) = \bigcup_{\boldsymbol{u} \in \mathbf{V}_{\mathcal{V}^{\text{ind}}}} \big\{ v \mapsto i \in \mathrm{I} : \mathrm{I} = \mathrm{M} + \mathbf{S} \cdot \big(\boldsymbol{u} - \mathsf{mid}\big(\mathbf{V}_{\mathcal{V}^{\text{ind}}}\big)\big) \big\} \tag{13}$$

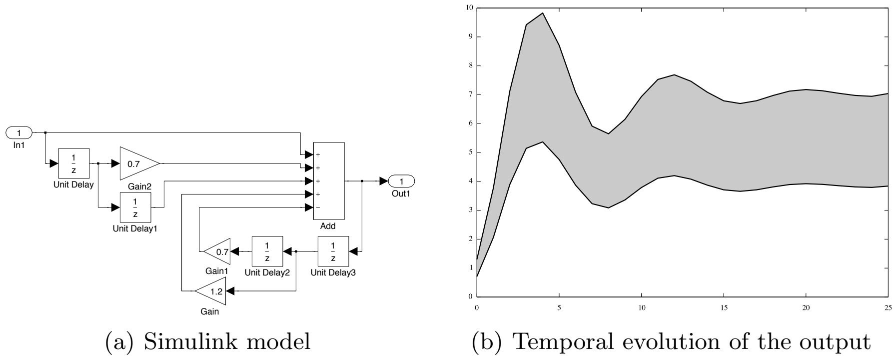
In Theorem 1, we state the soundness of the floating-point analysis using FPS domain with respect to the concrete floating-point semantics. The later is based on the concrete semantics of floating-point expressions $[\![e]\!]$, see [24] for its definition.

Theorem 1. *If the set of concrete environments env is contained in the abstract environment $env^\sharp$ then we have for all instruction i representing either an assignment or a test:*

$$[\![i]\!](env) \subseteq \gamma_{\mathbb{S}}\left([\![i]\!]^\sharp(env^\sharp)\right) \; .$$

5 Case Studies

In this section, we present experimental results of the static analysis of numerical programs using our floating-point slope domain. We based our examples on Matlab/Simulink models which are block-diagrams. We present as examples a second order linear filter and a square root computation with a Newton method.



(a) Simulink model (b) Temporal evolution of the output

Fig. 3. Second order linear filter

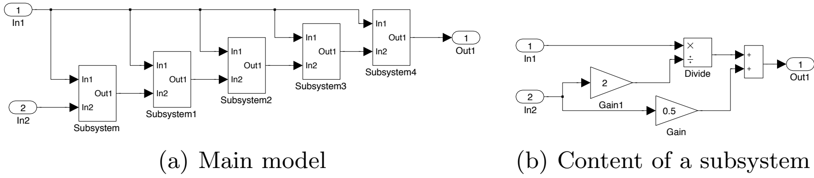
We first give a quick view of Matlab/Simulink models. In a block-diagram, each node represents an operation and each wire represents a value evolving during time. We consider a few operations such that arithmetic operations, gain operation that is multiplication by a constant, conditional statement (called *switch*[9] in Simulink), and *unit delay* block represented by $\frac{1}{\mathrm{z}}$ which acts as a memory. We can hence write discrete-time models thanks to finite difference equations, see [3] for further details.

The semantics of Simulink models is based on finite-time execution. In other words, a Simulink model is implicitly embedded in a *simulation loop* modelling the temporal evolution starting from $t = 0$ to a given final time t_{end}. The body of this loop follows three steps: *i)* evaluating the inputs, *ii)* computing the outputs, *iii)* updating the state variables *i.e.* values of the unit delay blocks. The static analysis of Simulink models transforms the simulation loop into a fixpoint computation. In its simple form, see [3] for further details, we add an extra time instant to collect all the behaviors from t_{end} to $t = +\infty$.

Linear Filter. We applied the floating-point slope domain on a second order linear filter defined by: $y_n = x_n + 0.7x_{n-1} + x_{n-2} + 1.2y_{n-1} - 0.7y_{n-2}$. The block-diagrams of this filter is given in Figure 3(a). We consider a simulation time of 25 seconds that is we unfold the simulation loop 25 times before making unions. The input belongs into the interval $[0.71, 1.35]$. The output of the filter is given in Figure 3(b). We consider, in this example, that $\mathcal{V}^{\mathrm{ind}}$ contains the input and the four unit delay blocks that is there are five independent variables. The gray area represents all the possible trajectories of the output corresponding of the set of inputs. Hence we can bound the output, without using the widening operator, by the interval $[0.7099, 9.8269]$.

Newton Method. We applied our domain on a Newton algorithm which computes the square root of a number a using the following iterative sequence:

[9] This operation is equivalent to the conditional expression: if $p_c(e_0)$ then e_1 else e_2. The predicate p_c has the form $e_0 \diamond c$ where c is a given constant and $\diamond \in \{\geq, >, \neq\}$.



(a) Main model (b) Content of a subsystem

Fig. 4. Simulink model of the square root computation

$x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n}$. We want to compute x_5 that is we consider the result of the Newton method after five iterations. The Simulink model is given in Figure 4(a) and in Figure 4(b), we give the model associated to one iteration of the algorithm. In this case, the set $\mathcal{V}^{\text{ind}}$ is only made of one element. For the interval input $[4, 8]$ with the initial value equals to 2, we have the result $[1.8547, 3.0442]$.

6 Related Work

Numerical domains have been intensively studied. A large part of numerical domains concern the polyhedral representation of sets. For example, we have the domain of polyhedron [10] and the variants [32,25,31,29,8,22,21,6,7]. We also have the numerical domains based on affine relations between variables [19,12] or the domain of linear congruences [16]. In general, all these domains are based on arithmetic with "good" properties such that rational numbers or real numbers. A notable exception is the floating-point versions of the octagon domain [24] and of the domain of polyhedron [5]. These domains give a sound over-approximation of the floating-point behaviors but they are not empowered to model the behaviors of floating-point arithmetic as we do.

Our FPS domain is more general than numerical abstract domains made for a special purpose. For example, we have the domain for linear filters [11] or for the numerical precision [14] which provide excellent results. Nevertheless as we showed in Section 5, we can apply this domain in various situations without losing too much precision.

7 Conclusion

We presented a new partially relational abstract numerical domain called FPS dedicated to floating-point variables. It is based on Krawczyk and Neumaier's work [20] on interval expansion of rational function using interval slopes. This domain is able to mimic the behaviors of the floating-point arithmetic such that the *absorption* phenomenon. We also presented experimental results showing the practical use of this domain in various contexts.

We want to pursue the work on the FPS domain by refining the the meet operation in order to keep relations between variables. Moreover we would like to model more closely the behaviors of floating point arithmetic, for example by taking into account the hardware instructions [26, Sect. 3].

As an other future work, we want to apply FPS domain for the analyses of the numerical precision by combining the FPS domain and domains defined in [23,4]. An interesting direction should be to make an analysis of the numerical precision by comparing results of the FPS domain and results coming from the other numerical domain which bound the exact mathematical behaviors such that [5]. Hence we can avoid the manipulation of complex abstract values to represent rounding errors such as in [23,14,4].

Acknowledgements. The author deeply thanks O. Bouissou, S. Graillat, T. Hilaire, D. Massé and M. Martel for their useful comments on the earlier versions of this article. He is also very grateful to anonymous referees who helped improving this work.

References

1. Bischof, C.H., Hovland, P.D., Norris, B.: Implementation of automatic differentiation tools. In: Partial Evaluation and Semantics-Based Program Manipulation, pp. 98–107. ACM Press, New York (2002)
2. Boldo, S., Nguyen, T.: Hardware-independant proofs of numerical programs. In: NASA Formal Methods Symposium (2010)
3. Chapoutot, A., Martel, M.: Abstract simulation: a static analysis of Simulink models. In: International Conference on Embedded Systems and Software, pp. 83–92. IEEE Press, Los Alamitos (2009)
4. Chapoutot, A., Martel, M.: Automatic differentiation and Taylor forms in static analysis of numerical programs. Technique et Science Informatiques 28(4), 503–531 (2009) (in French)
5. Chen, L., Miné, A., Patrick, C.: A sound floating-point polyhedra abstract domain. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 3–18. Springer, Heidelberg (2008)
6. Chen, L., Miné, A., Wang, J., Cousot, P.: Interval polyhedra: an abstract domain to infer interval linear relationships. In: Palsberg, J., Su, Z. (eds.) Static Analysis. LNCS, vol. 5673, pp. 309–325. Springer, Heidelberg (2009)
7. Chen, L., Miné, A., Wang, J., Cousot, P.: An abstract domain to discover interval linear equalities. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 112–128. Springer, Heidelberg (2010)
8. Clarisó, R., Cortadella, J.: The Octahedron abstract domain. Science Computer Programming 64(1), 115–139 (2007)
9. Cousot, P., Cousot, R.: Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252. ACM, New York (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages, pp. 84–97. ACM Press, New York (1978)
11. Férêt, J.: Static analysis of digital filter. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004)
12. Ghorbal, K., Goubault, E., Putot, S.: The zonotope abstract domain Taylor1+. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 627–633. Springer, Heidelberg (2009)

13. Goubault, E.: Static analyses of floating-point operations. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 234–259. Springer, Heidelberg (2001)
14. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 18–34. Springer, Heidelberg (2006)
15. Granger, P.: Improving the results of static analyses programs by local decreasing iteration. In: Shyamasundar, R.K. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 68–79. Springer, Heidelberg (1992)
16. Granger, P.: Static analysis of linear congruence equalities among variables of a program. In: Abramsky, S. (ed.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493, pp. 169–192. Springer, Heidelberg (1991)
17. Higham, N.: Accuracy and stability of numerical algorithms, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2002)
18. IEEE Task P754: IEEE 754-2008, Standard for Floating-Point Arithmetic. Institute of Electrical, and Electronic Engineers (2008)
19. Karr, M.: Affine relationships among variables of a program. Acta Informatica 6, 133–151 (1976)
20. Krawczyk, R., Neumaier, A.: Interval slopes for rational functions and associated centered forms. SIAM Journal on Numerical Analysis 22(3), 604–616 (1985)
21. Laviron, V., Logozzo, F.: Subpolyhedra: a (more) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
22. Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: Symposium on Applied Computing, pp. 184–188. ACM, New York (2008)
23. Martel, M.: Semantics of roundoff error propagation in finite precision computations. Higher Order and Symbolic Computation 19(1), 7–30 (2004)
24. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
25. Miné, A.: The Octagon abstract domain. Journal of Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
26. Monniaux, D.: Compositional analysis of floating-point linear numerical filters. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 199–212. Springer, Heidelberg (2005)
27. Moore, R.: Interval analysis. Prentice-Hall, Englewood Cliffs (1966)
28. Muller, J.M., Brisebarre, N., De Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of floating-point arithmetic. Birkhauser, Boston (2009)
29. Péron, M., Halbwachs, N.: An abstract domain extending difference-bound matrices with disequality constraints. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 268–282. Springer, Heidelberg (2007)
30. Rump, S.: Expansion and estimation of the range of nonlinear functions. Mathematics of Computation 65(216), 1503–1512 (1996)
31. Sankaranarayanan, S., Colon, M., Sipma, H., Manna, Z.: Efficient strongly relational polyhedral analysis. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 111–125. Springer, Heidelberg (2005)
32. Simon, A., King, A., Howe, J.: Two variables per linear inequality as an abstract domain. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)

A Shape Analysis for Non-linear Data Structures

Renato Cherini, Lucas Rearte, and Javier Blanco

FaMAF, Universidad Nacional de Córdoba, 5000 Córdoba, Argentina

Abstract. We present a terminating shape analysis based on Separation Logic for programs that manipulate non-linear data structures such as trees and graphs. The analysis automatically calculates concise invariants for loops, with a level of precision depending on the manipulations applied on each program variable. We report experimental results obtained from running a prototype that implements our analysis on a variety of examples.

1 Introduction

Shape Analysis is a form of static code analysis for imperative programs using dynamic memory that attempts to discover and verify properties of linked data structures such as lists, trees, heaps, etc. A shape analysis is intended not only to report null-pointer dereferences or memory leaks, but also to analyze the validity of non trivial properties about the shape of the dynamic structures in memory.

There exists in the literature a variety of shape analysis based on different theories, for instance: 3-valued logic ([22,28]), monadic second order logic ([24,20]), and Separation Logic ([12,4,25,11]). In particular the latter received recently great attention due to the advantages of local reasoning enabled by their logical framework ([27,26]), which leads to modular analysis that can be relatively easily extended to support large-scale programs ([7,3,6,33,15]), concurrent programming ([16,31,9,8]) and object-oriented programming ([13,29]).

This family of shape analysis is based on a symbolic execution of the program over abstract states specified by formulæ of a restricted subset of Separation Logic, including predicates describing linear data structures, possibly combined in intricate ways, such as linked lists, doubly linked lists, etc. In this article we explore the possibility of extending the shape analysis of [12] to support non-linear data structures such as trees and graphs. At a first glance, we can foresee some difficulties:

- The use of a naive predicate to describe the structures, like the usual tree predicate of Separation Logic, would lead to a unmanageable proliferation of occurrences of predicates in formulæ, caused by the inherent multiple recursion. In the case of graphs, there is no well established predicate in the literature which adequately specifies them.
- Algorithms over trees and graphs are often more complex than their counterparts on lists. Usually they involve intensive pointer manipulation and nested control structures. Algorithms like the Schorr-Waite graph traversal

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 201–217, 2010.
© Springer-Verlag Berlin Heidelberg 2010

are usually proposed as challenges for any new verification methodology. Their complexity imposes strong requirements for precision in calculating the loop invariants needed to verify interesting properties.
– Every particular path in the traverse of a structure is usually relevant to the validity of interesting properties satisfied by a given algorithm. However the multiple links of non-linear structure nodes triggers an exponential growth of the number of paths in traversing such structures. This raises the need for a balance between precision and abstraction to prevent an excessive growth in the number of formulæ composing an abstract state.

The contribution of this article is a terminating shape analysis which, given a precondition for a program, automatically computes a postcondition and invariants for each program loop. This analysis adjusts the level of abstraction in the computation of invariants, taking into account the information requirements for the the manipulation applied to each variable. Thus, the calculated invariants turn out to be compact and accurate enough in most practical cases. In order to define the analysis we have introduced a linear recursive predicate for the description of (families of) binary trees, leading to simple specifications of partial data structures ocurring at intermediate points in the execution of an iterative algorithm. This predicate can be easily adapted to deal with other data structures, such as threaded trees, balanced trees, graphs, etc.

The article is organized as follows. In section 2 we adapt the semantic setting from [12] to our purposes. First, we introduce the concrete memory model and semantics that defines the programming language for algorithms manipulating binary trees. Then we present an executable intermediate semantics that considers programs as abstract state transformers. Each abstract state consists of a set of restricted formulæ of Separation Logic called Symbolic Heaps, representing all concrete states which satisfy any of these formulæ. In section 3 we extend this semantics by introducing an abstraction phase for computing loop invariants, defining an executable and terminating abstract semantics. In section 4 we briefly present an extension of our framework to the domain of graphs. In section 5 we show the experimental results of our analysis on a variety of examples. Finally, in section 6 we discuss the conclusions and related work.

2 Semantic Settings

2.1 Concrete Semantics

We assume the existence of a *finite* set Vars of program variables $x, y, \ldots$, with values in $\mathsf{Locations} \cup \{\mathbf{nil}\}$. A state is a pair consisting of a (total) function Stack from program variables to values, and a partial function Heap mapping allocated memory adresses to triples of values (l, r, v), representing a binary tree node with value v and links to left and right subtrees l and r respectively.

$$\mathsf{Values} \supseteq \mathsf{Locations} \cup \{\mathbf{nil}\} \qquad \mathsf{Heaps} \doteq \mathsf{Locations} \rightharpoonup_f (\mathsf{Values}, \mathsf{Values}, \mathsf{Values})$$
$$\mathsf{Stacks} \doteq \mathsf{Vars} \to \mathsf{Values} \qquad \mathsf{States} \doteq \mathsf{Stacks} \times \mathsf{Heaps}$$

$$[x]_e.s \doteq s.x \qquad\qquad [\mathbf{nil}]_e.s \doteq \mathbf{nil}$$

$$\frac{[e]_e.s = v}{(s,h),\ x := e \ \rightsquigarrow\ (s|x \to v, h)} \qquad \frac{l \notin dom.h}{(s,h),\ \mathbf{new}(x) \ \rightsquigarrow\ (s|x \to l, h|l \to (v_1, v_2, v_3))}$$

$$\frac{[y]_e.s = l \quad h.l.i = v}{(s,h),\ x := y.i \ \rightsquigarrow\ (s|x \to v, h)} \qquad \frac{[x]_e.s = l \quad l \in dom.h}{(s,h),\ \mathbf{free}(x) \ \rightsquigarrow\ (s, h - l)}$$

$$\frac{[y]_e.s = l \quad [e]_e.s = v \quad l \in dom.h}{(s,h),\ y.i := e \ \rightsquigarrow\ (s, h|l.i \to v)} \qquad \frac{[x]_e.s \notin dom.h}{(s,h),\ a(x) \ \rightsquigarrow\ \top}$$

Fig. 1. Concrete semantics of expressions and atomic commands

We use a simple imperative programming language with explicit heap manipulation commands for lookup, mutation, allocation and disposing, given by the following grammar, where x is a program variable, and $0 \leq i \leq 2$:

$e ::= x \mid \mathbf{nil} \mid 0 \mid 1 \ldots$	Expressions
$eq ::= e = e \mid e \neq e$	(In)Equalities
$b ::= (eq \wedge \ldots \wedge eq) \vee \ldots \vee (eq \wedge \ldots \wedge eq)$	DNF Boolean Expressions
$a ::= x := e \mid x := x.i \mid x.i := e \mid \mathbf{new}(x) \mid \mathbf{free}(x)$	Atomic Commands
$p ::= a \mid p;p \mid \mathbf{while}\ b\ \mathbf{do}\ p \mid \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ p$	Compound Commands

The concrete semantics is given by continuous functions $[\![p]\!]_c : \mathsf{D}_c \to \mathsf{D}_c$ for each program p on the complete lattice D_c defined as the topped powerset of States, i.e. $\mathsf{D}_c \doteq \mathcal{P}(\mathsf{States} \cup \{\top\})$. The distinguished element $\top$ represents an execution that terminates abnormally with a memory fault. The order of the lattice is given by set inclusion $\subseteq$, taking $\top$ as the top element and equating all sets that contain it.

For each atomic command a we present a relation $\rightsquigarrow \subseteq \mathsf{State} \times \mathsf{State} \cup \{\top\}$ in figure 1. *Notation: We use $a(x)$ to denote an atomic command that accesses the heap cell x, period (.) for function application, $f|x \to v$ for the update of function at x with new value v, $dom.f$ and $img.f$ for the domain and the image of a function f respectively, and $f - x$ for the elimination of x from the partial function f.* The relation $\rightsquigarrow$ mimics the standard operational semantics for Separation Logic [27], and can be easily extended to a function $a^{\restriction} : \mathsf{D}_c \to \mathsf{D}_c$:

$$a^{\restriction}.S \doteq \{\sigma' \mid \exists \sigma \in S \cdot \sigma, a \ \rightsquigarrow\ \sigma' \text{ or } (\sigma = \sigma' = \top)\}$$

Thus concrete semantics for atomic command a is defined as $[\![a]\!]_c = a^{\restriction}$. This semantics is extended to compound commands as usual:

$$\begin{aligned} [\![p;p']\!]_c &\doteq [\![p']\!]_c \circ [\![p]\!]_c \\ [\![\mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ p']\!]_c &\doteq ([\![p]\!]_c \circ \mathsf{filter}.b) \cup ([\![p']\!]_c \circ \mathsf{filter}.\neg b) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ p]\!]_c &\doteq \lambda S \cdot \mathsf{filter}.\neg b \circ (\mu S' \cdot S \cup ([\![p]\!]_c \circ \mathsf{filter}.b).S') \end{aligned}$$

where we use μ to denote the minimal fixed point operator, $\circ$ the functional composition, and $\neg$ a meta-operation that transforms a boolean expression b in its negation in disjunctive normal form. Function filter removes states that are inconsistent with the truth value of boolean expression b of guarded commands.

$s \models \mathbf{true}$	always
$s \models eq$	iff $[eq]_e.s$
$s \models \Pi_1 \wedge \Pi_2$	iff $s \models \Pi_1$ and $s \models \Pi_2$
$(s,h) \models \mathbf{emp}$	iff $h = \emptyset$
$(s,h) \models \mathbf{junk}$	iff $h \neq \emptyset$
$(s,h) \models \mathbf{true}$	always
$(s,h) \models x \mapsto l,r,v$	iff $h = \{([x]_e.s, ([l]_e.s, [r]_e.s, [v]_e.s))\}$
$(s,h) \models \Sigma_1 * \Sigma_2$	iff there exist h_1, h_2 such that $h_1 \cap h_2 = \emptyset$ and $h = h_1 \cup h_2$ and $(s,h_1) \models \Sigma_1$ and $(s,h_2) \models \Sigma_2$
$(s,h) \models \Pi \mid \Sigma$	iff there exists $\overline{v}'$ such that $s\vert\overline{x}' \to \overline{v}' \models \Pi$ and $(s\vert\overline{x}' \to \overline{v}', h) \models \Sigma$ where $\overline{x}'$ is the (sequence of) primed variable(s) in $\Pi \mid \Sigma$ and $\overline{v}'$ is a (sequence of) fresh value(s).

Fig. 2. Semantics of Symbolic Heaps

2.2 Symbolic Heaps and Intermediate Semantics

In order to define intermediate semantics, it is necessary to extend the concrete state model, assuming the existence of an infinite set Vars' of primed variables. These variables are implicitly existentially quantified in the semantics and are intended to be used only in formulæ and not within the program text.

A symbolic heap $\Pi \mid \Sigma$ consists of a formula Π of *pure* predicates about equalities and inequalities of variables, and a formula Σ of *spatial* predicates about the heap, according to the following grammar:

$e ::= x \mid x' \mid \mathbf{nil} \mid 0 \mid 1 \ldots$	Expressions
$ms ::= \{\!\{e, e, \ldots, e\}\!\}$	Multiset Expressions
$\Pi ::= \mathbf{true} \mid eq \mid \Pi \wedge \Pi$	Pure Predicates
$\Sigma ::= \mathbf{emp} \mid \mathbf{true} \mid \mathbf{junk} \mid v \mapsto e,e,e \mid \mathbf{trees}.ms.ms \mid \Sigma * \Sigma$	Heap Predicates
$SH ::= \Pi \mid \Sigma$	Symbolic Heaps

where $x \in$ Vars, $x' \in$ Vars', $v \in$ Vars $\cup$ Vars', eq is defined as in the previous section, and the expressions ms represent constant multisets of expressions. *Notation: we use metavariables $\mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \ldots$ to denote multisets, $\uplus$ for the sum of multisets, $\emptyset$ for the empty multiset, $\oplus$ for insertion of an item instance, $\ominus$ for deletion of all item instances, $\Subset$ for membership, and $\not\Subset$ for non-membership.*

As symbolic heaps represent a fragment of Separation Logic, its semantics with respect to concrete state model is mostly standard. We present it in figure 2, except for **trees** predicate. We briefly discuss the spatial component; interested reader should refer to [27] for more details. The predicate **emp** specifies that no dynamic memory is allocated, whereas **junk** states there is garbage, consisting in some allocated but inaccessible cell. The predicate **true** is valid in any heap and it will be heavily used to indicate the *possible* existence of garbage. 'Points-to' predicate $x \mapsto l,r,v$ specifies the heap consisting of a single memory cell with address x and value (l,r,v). The *spatial conjunction* $\Sigma_1 * \Sigma_2$ is valid if the heap can be divided into two disjoint subheaps satisfying Σ_1 and Σ_2 respectively.

The predicate **trees** is intended to define (a family of) binary trees. More generally, $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ defines a family of possible partial trees with roots in $\mathcal{C}$ and dangling pointers in $\mathcal{D}$. On the one hand, a dangling pointer of a partial tree can point to an internal node of another tree. On the other hand, a dangling pointer can be shared by two or more partial trees. Thus, $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ defines a binary node structure which allows the possibility of sharing, but only restricted to the heap cells mentioned by $\mathcal{D}$. More precisely, $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ is defined by the least predicate satisfying the following equations of the standard Separation Logic:

$$\begin{aligned}
\mathbf{trees}.\emptyset.\emptyset &\doteq \mathbf{emp}\\
\mathbf{trees}.\emptyset.\mathbf{nil} \oplus \mathcal{D} &\doteq \mathbf{trees}.\emptyset.\mathcal{D}\\
\mathbf{trees}.x \oplus \mathcal{C}.\mathcal{D} &\doteq ((x = \mathbf{nil} \vee x \Subset \mathcal{D}) \wedge \mathbf{trees}.\mathcal{C}.(x \ominus \mathcal{D})) \vee\\
& ((x \neq \mathbf{nil} \wedge x \not\Subset \mathcal{D}) \wedge (\exists l, r, v \cdot x \mapsto l, r, v * \mathbf{trees}.l \oplus r \oplus \mathcal{C}.\mathcal{D}))
\end{aligned}$$

The predicate **trees** is useful to specify the intermediate structures that occur during loop iteration and it satisfies some nice syntactic properties. The possible sharing is manageable due to the kind of formulæ that usually specify the preconditions of interest. The relation to the standard predicate **tree**[1] occurring in the literature of Separation Logic, is quite straightforward:

Lemma 1. *The following is a valid formula in Separation Logic:*

$$\mathbf{trees}.\{\!|x, y, \ldots, z|\!\}.\emptyset \Leftrightarrow \mathbf{tree}.x * \mathbf{tree}.y * \ldots * \mathbf{tree}.z$$

Analogously to the concrete semantics, the intermediate semantics is defined on the lattice $\mathsf{D}_a \doteq \mathcal{P}(\mathsf{SH} \cup \top)$, where SH denotes the set of symbolic heaps. For each atomic command a transition relation $\leadsto\ \subseteq \mathsf{SH} \times \mathsf{SH} \cup \{\top\}$ is defined:

$$\Pi \,\dot{|}\, \Sigma,\ x := e \ \leadsto\ \Pi' \wedge x = e_{/x \leftarrow x'} \,\dot{|}\, \Sigma' \qquad \Pi \,\dot{|}\, \Sigma,\ \mathbf{new}(x) \ \leadsto\ \Pi' \,\dot{|}\, \Sigma' * x \mapsto \overline{e'}$$

$$\frac{\overline{g} = \overline{e}|i \rightarrow f}{\Pi \,\dot{|}\, \Sigma * x \mapsto \overline{e},\ x.i := f \ \leadsto\ \Pi \,\dot{|}\, \Sigma * x \mapsto \overline{g}} \qquad \Pi \,\dot{|}\, \Sigma * x \mapsto \overline{e},\ \mathbf{free}(x) \ \leadsto\ \Pi \,\dot{|}\, \Sigma$$

$$\frac{f = \overline{e}.i_{/x \leftarrow x'}}{\Pi \,\dot{|}\, \Sigma * y \mapsto \overline{e}, x := y.i \ \leadsto\ \Pi' \wedge x = f \,\dot{|}\, \Sigma' * (y \mapsto \overline{e})_{/x \leftarrow x'}}$$

Notation: we denote by $\overline{e}$ *a triple of expressions, by* $\overline{e'}$ *a triple of quantified fresh variables, and by* $P_{/x \leftarrow y}$ *the syntactic substitution of* y *for* x *in* P*. In every case* $\Pi' = \Pi_{/x \leftarrow x'}$*, and* $\Sigma' = \Sigma_{/x \leftarrow x'}$*, where* $x' \in \mathit{Vars}'$ *is a fresh variable.*

Commands for mutation, lookup and disposing require that the pre-state explicitly indicates the existence of the cell to be dereferenced. To ensure this, we define a function $\mathsf{rearr}.x : \mathsf{D}_a \to \mathsf{D}_a$ that given a variable of interest x tries to reveal the memory cell pointed to by x in every symbolic heap in certain abstract

[1] This is defined as the least predicate that satisfies the equation

$$\mathbf{tree}.x \Leftrightarrow (x = \mathbf{nil} \wedge \mathbf{emp}) \vee (\exists l, r, v \cdot x \mapsto l, r, v * \mathbf{tree}.l * \mathbf{tree}.r)$$

$$
\begin{array}{ll}
\Pi \,\vert\, \Sigma \vdash x = \mathbf{nil} & \text{if } (x \equiv \mathbf{nil}) \text{ or } (\Pi = \Pi' \wedge x = y \text{ and } \Pi' \,\vert\, \Sigma \vdash y = \mathbf{nil}) \\
\Pi \,\vert\, \Sigma \vdash x = y & \text{if } (x \equiv y) \text{ or } (\Pi = \Pi' \wedge x = z \text{ and } \Pi' \,\vert\, \Sigma \vdash z = y) \\
\Pi \,\vert\, \Sigma \vdash x \neq \mathbf{nil} & \text{if } (\Pi = \Pi' \wedge x \neq \mathbf{nil}) \text{ or } (\Sigma = \Sigma' * y \mapsto \overline{e} \text{ and } \Pi \,\vert\, \Sigma \vdash x = y) \\
 & \text{or } (\Pi = \Pi' \wedge z = \mathbf{nil} \text{ and } \Pi' \,\vert\, \Sigma \vdash x \neq z) \\
 & \text{or } (\Pi = \Pi' \wedge x = y \text{ and } \Pi' \,\vert\, \Sigma \vdash y \neq \mathbf{nil}) \\
\Pi \,\vert\, \Sigma \vdash x \neq y & \text{if } (\Pi = \Pi' \wedge x \neq y) \text{ or } (\Pi = \Pi' \wedge x = z \text{ and } \Pi' \,\vert\, \Sigma \vdash z \neq y) \\
 & \text{or } (\Pi \,\vert\, \Sigma \vdash \mathit{Cell}(x) * \mathit{Cell}(y) \text{ and } \Pi \,\vert\, \Sigma \vdash x \neq \mathbf{nil} \vee y \neq \mathbf{nil}) \\
\Pi \,\vert\, \Sigma \vdash x \in \emptyset & \text{never} \\
\Pi \,\vert\, \Sigma \vdash x \in \{\!|e_1, .., e_n|\!\} & \text{if } \Pi \,\vert\, \Sigma \vdash x = e_i \text{ for some } i \text{ such that } 1 \leq i \leq n \\
\Pi \,\vert\, \Sigma \vdash x \notin \emptyset & \text{always} \\
\Pi \,\vert\, \Sigma \vdash x \notin \{\!|e_1, .., e_n|\!\} & \text{if } \Pi \,\vert\, \Sigma \vdash x \neq e_i \text{ for every } i \text{ such that } 1 \leq i \leq n \\
\Pi \,\vert\, \Sigma \vdash eq_1 \wedge \ldots \wedge eq_n & \text{if } \Pi \,\vert\, \Sigma \vdash eq_1 \text{ and} \ldots \text{and } \Pi \,\vert\, \Sigma \vdash eq_n \\
\Pi \,\vert\, \Sigma \vdash eq_1 \vee \ldots \vee eq_n & \text{if } \Pi \,\vert\, \Sigma \vdash eq_1 \text{ or} \ldots \text{or } \Pi \,\vert\, \Sigma \vdash eq_n \\
\Pi \,\vert\, \Sigma \vdash \mathit{False} & \text{if } (\Sigma = \Sigma' * x \mapsto \overline{e} \text{ and } \Pi \,\vert\, \Sigma \vdash x = \mathbf{nil}) \\
 & \text{or } (\Sigma = \Sigma' * P(x) * P(y) \text{ and } \Pi \,\vert\, \Sigma \vdash x = y \wedge x \neq \mathbf{nil}) \\
\Pi \,\vert\, \Sigma \vdash \mathit{Cell}(x_1) * \ldots * \mathit{Cell}(x_n) & \text{if } \Sigma = \Sigma' * P(y_1) * \ldots * P(y_n) \\
 & \text{and } \Pi \,\vert\, \Sigma' \vdash x_1 = y_1 \wedge \ldots \wedge x_n = y_n \\
\multicolumn{2}{c}{\text{where } P(x) \equiv x \mapsto \overline{e} \text{ or } (P(x) \equiv \mathbf{trees}.x \oplus \mathcal{C}.\mathcal{D} \text{ and } \Pi \,\vert\, \Sigma' \vdash x \notin \mathcal{D})}
\end{array}
$$

Fig. 3. Syntactic Theorem Prover $\vdash$

state. Function $\mathsf{rearr}.x$ applies rewrite rules to every $\Pi \,\vert\, \Sigma$ until $x \mapsto \overline{e}$ occurs in Σ; otherwise it returns $\{\top\}$ when no rule can be applied.

Rewrite rules try to reveal a memory cell through equalities of variables and the unfolding of **trees** predicates:

$$\frac{\Pi \,\vert\, \Sigma \vdash x = y}{\Pi \,\vert\, \Sigma * y \mapsto \overline{e} \implies \Pi \,\vert\, \Sigma * x \mapsto \overline{e}}\,[\mathsf{Eq}]$$

$$\frac{\Pi \,\vert\, \Sigma \vdash x = y \wedge y \neq \mathbf{nil} \wedge y \notin \mathcal{D}}{\Pi \,\vert\, \Sigma * \mathbf{trees}.y \oplus \mathcal{C}.\mathcal{D} \implies \Pi \,\vert\, \Sigma * x \mapsto l', r', v' * \mathbf{trees}.l' \oplus r' \oplus \mathcal{C}.\mathcal{D}}\,[\mathsf{Unfold}]$$

where l', r', v' are fresh variables. The conditions for application of these rules require the ability to decide on the validity of certain predicates, denoted by $\vdash$. In figure 3 we present a small and simple syntactic theorem prover that in some circumstances allows to derive the validity of predicates like (in)equality of expressions, (non-)membership in a multiset, etc. *Notation: we use $\equiv$ for syntactic equality; x, y, z, x_i, y_i are variables in $\mathit{Var} \cup \mathit{Var}'$.*

Taking $_^{\upharpoonright}$ defined analogously as in the previous section, we define the intermediate semantics for an atomic command a as:

$$[\![a(x)]\!]_i \doteq a(x)^{\upharpoonright} \circ \mathsf{rearr}.x$$

The semantics of compound statements follows the same pattern as the concrete semantics, leaving only the function $\mathsf{filter}.b$ to be defined. Recall that a program guard b is a disjunction of terms eqs that are conjunctions of (in)equalities eq. If $b \doteq eqs_1 \vee \ldots \vee eqs_n$ we define the function $\mathsf{filter}.b$ as:

$$\mathsf{filter}.b.S \doteq \{\top \mid \top \in S\} \cup \bigcup_{1 \leq i \leq n} \{\Pi \wedge eqs_i \,\vert\, \Sigma \mid \Pi \,\vert\, \Sigma \in S \text{ and } \Pi \,\vert\, \Sigma \nvdash \neg eqs_i\}$$

Rewriting rules defining rearr represent valid semantic implications. But although the algorithm for $\vdash$ is consistent, clearly it is not complete. Therefore, on some circustamces rearr will not be able to reveal the existence of a particular memory cell with the consequence that certain executions of intermediate semantics finish in $\{\top\}$, even when there is no memory violation according to the concrete semantics. To establish this, we need to define a relation between both semantics using a concretization function $\gamma : \mathcal{P}(\mathsf{SH} \cup \{\top\}) \to \mathcal{P}(\mathsf{States} \cup \{\top\})$:

$$\gamma.S \doteq \begin{cases} \{\top\} & \text{if } \top \in S \\ \{(s,h) \mid \exists \Pi \,\vert\, \Sigma \in S \cdot (s,h) \models \Pi \,\vert\, \Sigma\} & \text{otherwise} \end{cases}$$

Extending the semantics $\models$ for any predicate P of figure 3 in the obvious way, we can prove the following results.

Lemma 2. (Soundness of $\Longrightarrow$)

1. *If $\Pi \,\vert\, \Sigma \vdash P$, then $(s,h) \models \Pi \,\vert\, \Sigma$ implies $(s,h) \models P$ for all s,h.*
2. *If $\Pi \,\vert\, \Sigma \Longrightarrow \Pi' \,\vert\, \Sigma'$, then $(s,h) \models \Pi \,\vert\, \Sigma$ implies $(s,h) \models \Pi' \,\vert\, \Sigma'$ for all s,h.*

Theorem 3. *The intermediate semantics is a sound over-approximation of the concrete semantics: $[\![p]\!]_c.(\gamma.S) \subseteq \gamma.([\![p]\!]_i.S)$ for all $S \in \mathcal{P}(\mathsf{SH} \cup \{\top\})$.*

3 Abstract Semantics: The Analysis

The intermediate semantics is executable but has no mechanism to facilitate the calculation of loop invariants and ensure termination. Generally, the execution of a loop dereferencing a variable x generates states with a large number of formulæ which contain an arbitrary number of terms of the form $x' \mapsto l', r', v'$. For example, the execution of the intermediate semantics on an algorithm that iteratively uses variable p to run through a binary search tree (BST) on the left link searching for its lowest value, starting from the expected precondition $\{\mathbf{true} \,\vert\, \mathbf{trees}.\{\!|x|\!\}.\emptyset\}$, generates statements of the form:

$$\begin{aligned}
&\{p = x \,\vert\, \mathbf{trees}.\{\!|x|\!\}.\emptyset,\\
&\ \mathbf{true} \,\vert\, x \mapsto p, r_1', v_1' * \mathbf{trees}.\{\!|p, r_1'|\!\}.\emptyset,\\
&\ \mathbf{true} \,\vert\, x \mapsto l_1', r_1', v_1' * l_1' \mapsto p, r_2', v_2' * \mathbf{trees}.\{\!|p, r_1', r_2'|\!\}.\emptyset,\\
&\ \mathbf{true} \,\vert\, x \mapsto l_1', r_1', v_1' * l_1' \mapsto l_2', r_2', v_2' * l_2' \mapsto p, r_3', v_3' * \mathbf{trees}.\{\!|p, r_1', r_2', r_3'|\!\}.\emptyset, \ldots
\end{aligned}$$

To handle this situation, we define an abstraction function $\mathsf{abs} : \mathsf{D}_a \to \mathsf{D}_a$, which aims at simplifying the formulæ of a state, replacing concrete information about the shape of the heap by a more abstract but still useful one. The use of such function is intended to facilitate the convergence of fixed-point computation which represents the semantics of a loop. Our abstract semantics $[\![\ ]\!]_a$ applies the function abs at the entry point and after each iteration of a loop. More precisely, the only change with respect to the intermediate semantics is:

$$[\![\mathbf{while}\ b\ \mathbf{do}\ p]\!]_a \doteq (\lambda S \cdot \mathsf{filter}.\neg b \circ (\mu S' \cdot \mathsf{abs}.(S \cup ([\![p]\!]_a \circ \mathsf{filter}.b).S'))) \circ \mathsf{abs}$$

$$\frac{\Sigma \doteq \Sigma' * \mathbf{trees}.\mathcal{C}.x \oplus \mathcal{D} * \mathbf{trees}.y \oplus \mathcal{E}.\emptyset \qquad \Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.\mathcal{C}.\mathcal{D} * \mathbf{trees}.\mathcal{E}.\emptyset} \textsf{[AbsTree1]}$$

$$\frac{\Pi \mid \Sigma \vdash x = y \qquad \Pi \mid \Sigma \vdash Cell(e) \vee e = \mathbf{nil} \quad \text{for all } e \Subset \mathcal{F}}{\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.x \oplus \mathcal{D} * T.y \oplus \mathcal{E}.\mathcal{F} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C} \uplus \mathcal{E}.\mathcal{D} \uplus \mathcal{F}} \textsf{[AbsTree2]}$$

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathbf{nil} \ominus \mathcal{C}.\mathbf{nil} \ominus \mathcal{D} \ \textsf{[AbsTree3]}$$

$$\frac{\Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma * \mathbf{trees}.x \oplus \mathcal{C}.y \oplus \mathcal{D} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D}} \textsf{[AbsTree4]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * \mathbf{trees}.y \oplus \mathcal{C}.\emptyset \qquad \Pi \mid \Sigma \vdash l = y \wedge x \neq r}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.\{\!|x|\!\}.\{\!|r|\!\} * \mathbf{trees}.\mathcal{C}.\emptyset} \textsf{[AbsArrow1]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * T.y \oplus \mathcal{C}.\mathcal{D} \qquad \Pi \mid \Sigma \vdash l = y \wedge x \not\Subset r \oplus \mathcal{D} \wedge y \not\Subset \mathcal{D}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.x \oplus \mathcal{C}.r \oplus \mathcal{D}} \textsf{[AbsArrow2]}$$

Fig. 4. Abstraction rules (first stage)

In this way, although the domain of abstract semantics remains $\mathcal{P}(\mathsf{SH} \cup \top)$, the semantics of a cycle is calculated over a subset of states which, as we will see in the following sections, ensure the convergence of fixed-point calculation.

The function abs is given by a set of rewriting rules. The more interesting ones are presented in Figure 4. *Notation: we use $T.\mathcal{C}.\mathcal{D}$ to denote either a term* $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ *or a term $x \mapsto l, r, v$ with $\mathcal{C} = \{\!|x|\!\}$ and $\mathcal{D} = \{\!|l, r|\!\}$.* The rules AbsArrow abstract predicates 'points-to' into predicates **trees**, forgetting the number of nodes that form the tree-like structure. The rules are presented for the left link being completely analogous for the right one. The rules AbsTree1-2 combine trees forgetting intermediate points between them. The conditions of application prevent the formation of cycles within the structure thus preserving the tree-like characteristic. The rules AbsTrees3-4 remove entry and outlet points which do not provide relevant information about the heap.

Recalling the example above, if we apply rules AbsArrow1, AbsTree2 and AbsTree4 in the second iteration of the fixed point calculation, we obtain the invariant:

$$\{p = x \mid \mathbf{trees}.\{\!|x|\!\}.\emptyset, \quad \mathbf{true} \mid \mathbf{trees}.\{\!|x|\!\}.\{\!|p|\!\} * \mathbf{trees}.\{\!|p|\!\}.\emptyset\}$$

3.1 Relevance of Variables and Abstraction

We begin by defining some terminology. We say that two terms $T_1.\mathcal{C}.\mathcal{D}$ and $T_2.\mathcal{E}.\mathcal{F}$ of a symbolic heap form a *chain link* if there exists an $x \Subset \mathcal{D}$ such that $x \Subset \mathcal{E}$. A sequence of terms $T_1, T_2, \ldots, T_n$ is a *chain* if T_i and T_{i+1} form a chain link for all $1 \leq i < n$. The application of abstraction rules involves losing two kinds of information about the heap: first, specific information on a cell referenced by some variable; second, the information about the variable that defines

a link between two terms. For example both cases ocurr in rule AbsArrow1, for x and y respectively. If an abstracted variable x (or y) is dereferenced at a later point of execution, this data loss can result in the impossibility of continuing a non-trivial analysis. This is either because it is not possible to reveal the cell pointed to by x, since application conditions for rearr rules are not granted any-more; or because the trace of the variable y as the midpoint of the tree-like structure is lost.

In general, the shape analysis derived from Separation Logic apply abstraction rules when the variable defining a link is quantified, but a similar approach is not adequate for our case. On the one side, it could become too strong as we have discussed before. On the other hand, it could be too weak, resulting in unnecessarily precise formulæ and therefore larger invariants. Keeping track of a chain of two, three or more links do not seem a problem in the case of linear structures. But in treating multilinked structures, given the many possible combinations, the number of formulæ needed to describe this chain grows exponentially. Although from the standpoint of correctness this is not a problem, to keep reduced abstract states speeds up the analysis and enables better understanding of the obtained invariants and postconditions.

The application of our abstraction rules is relative to a *relevance level* assigned to each variable. The level of a variable at a certain point of execution depends on the kind of commands involving it, which are intended to be executed after this point.

The very simple static analysis relev is performed on a program, and returns a function $f : \mathsf{Var} \rightarrow \mathsf{Nat}_0$ which assigns a value to each variable, encoding the foreseen needed information for it. Considering a program as a semicolon separated sequence of commands, relev updates a function f initialized in 0 for each variable, according to:

$$
\begin{aligned}
\mathsf{relev}.f.(x := e; ps) &= (\mathsf{relev}.f.ps \,|\, x \rightarrow 0){\uparrow}y \rightarrow 2 \quad \text{forall } y \in e \\
\mathsf{relev}.f.(x := y.i; ps) &= (\mathsf{relev}.f.ps \,|\, x \rightarrow 0){\uparrow}y \rightarrow 3 \\
\mathsf{relev}.f.(x.i := e; ps) &= (\mathsf{relev}.f.ps{\uparrow}x \rightarrow 4){\uparrow}y \rightarrow 2 \quad \text{forall } y \in e \\
\mathsf{relev}.f.(\mathbf{free}(x); ps) &= \mathsf{relev}.f.ps{\uparrow}x \rightarrow 3 \\
\mathsf{relev}.f.(\mathbf{new}(x); ps) &= \mathsf{relev}.f.ps \,|\, x \rightarrow 0 \\
\mathsf{relev}.f.(\mathbf{if}\ b\ \mathbf{then}\ ps_1 \mathbf{else}\ ps_2; ps) &= \\
&(\mathsf{relev}.f.(ps_1 +\!\!+\, ps) \ \max\ \mathsf{relev}.f.(ps_2 +\!\!+\, ps)){\uparrow}y \rightarrow 2 \quad \text{forall } y \in b \\
\mathsf{relev}.f.(\mathbf{while}\ b\ \mathbf{do}\ ps_1; ps) &= \\
&(\mathsf{relev}.f.(ps_1 +\!\!+\, ps) \ \max\ \mathsf{relev}.f.ps){\uparrow}y \rightarrow 2 \quad \text{forall } y \in b \\
\mathsf{relev}.f.\epsilon &= f
\end{aligned}
$$

Notation: $f \,|\, x \rightarrow n$ is the function update previously presented, ϵ is the empty sequence, $+\!\!+$ denotes concatenation, and operators $\uparrow$ and max *are defined as:*

$$
(f{\uparrow}y \rightarrow n).x \doteq \begin{cases} f.x & \text{if } x \neq y \text{ or } f.x \geq n \\ n & \text{otherwise} \end{cases} \qquad (f \max g).x \doteq \begin{cases} f.x & \text{if } f.x \geq g.x \\ g.x & \text{otherwise} \end{cases}
$$

Thus, the relevance level of x in $\Pi \,\text{¦}\, \Sigma$ is given by the highest value according to f within its equivalence class induced by terms $x = y$ in Π. Level 1 is reserved for variables of precondition and level -1 for quantified variables.

$\Pi \wedge x = e \,\dot{|}\, \Sigma \implies (\Pi \,\dot{|}\, \Sigma)_{/x \leftarrow e}$ [EqElim]
when relev computes for x a value < 1

$\Pi \wedge e_1 \neq e_2 \,\dot{|}\, \Sigma \implies \Pi \,\dot{|}\, \Sigma$ [NeqElim]
when no variable with relevance level greater than 0 appears in e_1, e_2

$$\Pi \,\dot{|}\, \Sigma * \mathbf{trees}.x' \oplus \mathcal{C}.\emptyset \implies \Pi \,\dot{|}\, \Sigma * \mathbf{trees}.\mathcal{C}.\emptyset * \mathbf{true} \text{ [Gb1]}$$

$$\Pi \,\dot{|}\, \Sigma * T.x' \oplus \mathcal{C}.\mathcal{D} \implies \Pi \,\dot{|}\, \Sigma * \mathbf{junk/true} \text{ [Gb2]}$$

$$\Pi \,\dot{|}\, \Sigma * \mathbf{trees}.\mathcal{C}.x' \oplus \mathcal{D} \implies \Pi \,\dot{|}\, \Sigma * \mathbf{true} \text{ [Gb3]}$$

$$\Pi \,\dot{|}\, \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D} * \mathbf{trees}.\mathcal{C}.\mathcal{D} \implies \Pi \,\dot{|}\, \Sigma \text{ [Gb4]}$$

$$\Pi \,\dot{|}\, \Sigma * x' \mapsto y', r, v * T.y' \oplus \mathcal{C}.\emptyset \implies \Pi \,\dot{|}\, \Sigma * x' \mapsto y', r, v * T.\mathcal{C}.\emptyset * \mathbf{true} \text{ [ChBrk1]}$$

$$\Pi \,\dot{|}\, \Sigma * x' \mapsto y', r, v * T.y' \oplus \mathcal{C}.\mathcal{D} \implies \Pi \,\dot{|}\, \Sigma * x' \mapsto y', r, v * \mathbf{junk/true} \text{ [ChBrk2]}$$

$$\Pi \,\dot{|}\, \Sigma * \mathbf{trees}.\mathcal{C}.y' \oplus \mathcal{D} * T.y' \oplus \mathcal{E}.\mathcal{F} \implies \Pi \,\dot{|}\, \Sigma * \mathbf{trees}.\mathcal{C}.y' \oplus \mathcal{D} * \mathbf{true} \text{ [ChBrk3]}$$

$$\Pi \wedge \mathbf{true} \,\dot{|}\, \Sigma \implies \Pi \,\dot{|}\, \Sigma \text{ [Zero1]} \qquad \Pi \,\dot{|}\, \Sigma * \mathbf{emp} \implies \Pi \,\dot{|}\, \Sigma \text{ [Zero2]}$$

$$\Pi \wedge eq \wedge eq \,\dot{|}\, \Sigma \implies \Pi \wedge eq \,\dot{|}\, \Sigma \text{ [Idemp1]}$$

$$\Pi \,\dot{|}\, \Sigma * \mathbf{true} * \mathbf{true} \implies \Pi \,\dot{|}\, \Sigma * \mathbf{true} \text{ [Idemp2]}$$

$$\Pi \,\dot{|}\, \Sigma * \mathbf{junk} * \mathbf{junk/true} \implies \Pi \,\dot{|}\, \Sigma * \mathbf{junk} \text{ [Junk]}$$

Fig. 5. Abstraction rules: second stage (top), third stage (bottom)

In our analysis, the relevance level required in each rule is parameterized, allowing us to adjust the abstraction performed. Possibly different requirements are impossed on variables being abstracted, depending on whether they define a link or they occur only as an address of a 'points-to' term. Abstracting variables with relevance level up to 2 in both cases, resulted in a safe choice to effectively carry out our experiments. Some algoritms allow higher limits, allowing more compact invariants and shorter execution times.

3.2 Quantified Variables and Termination

Quantified variables play a fundamental role in the completeness of the analysis induced by $[\![\ ]\!]_a$, since they can be used to generate an infinite set of formulæ as the image of abs. The way to generate these formulæ is by forming chains of terms T_i of arbitrary length, for which it is necessary to have an arbitrary large number of links formed with quantified variables. In circumstances where $img.\mathsf{abs}$ is infinite, the computation of the minimum fixed point that represents the semantics of a cycle cannot be carried out effectively.

However, abs consists of a number of rules that limit the number of quantified variables and ensures the finiteness of its image and therefore the convergence of fixed point calculation. The analysis is organized in three stages, each of which involve the exhaustive application of rewriting rules.

The first stage is defined by the abstraction rules previously presented (Fig. 4). If the limit imposed over the relevance level of variables to be abstracted is

greater than zero, the application of these rules thereby limit the length of the chains by removing a significant number of quantified variables, when it is possible to ensure the absence of cycles in the structure.

In a second stage, the top rules of Figure 5 are applied. *Notation: with* **junk**/**true** *we mean* **junk** *in the case that term* T *is a predicate 'points-to', and* **true** *if it is a predicate* **trees**. Essentially, these rules simplify the situations in which it is impossible to ensure a tree-like structure, and therefore the rules of the first stage do not apply. The rules ChBrk have the application condition $x', y' \notin \Sigma$ and prevent the formation of arbitrarily long chains by removing links formed with irrelevant quantified variables. Every Gb rule has the application condition $x' \notin \Pi \mid \Sigma$. The rules Gb1-2 remove all chains that do not start with program variables. Rules Gb3-4 bound chains starting with a term **trees**.$\mathcal{C}.\mathcal{D}$, either by eliminating terms containing quantified variables among its outlets, or by eliminating multiples ocurrences of a term, since they do not lead to an inconsistency only if $\mathcal{C} = \mathcal{D}$ and therefore **trees**.$\mathcal{C}.\mathcal{D} =$ **emp**.

Finally, in the third stage the bottom rules of Figure 5 are applied. They collect all garbage in one predicate **junk** or **true**, and delimit the number of pure formulæ (along with rules EqElim and NeqElim of the previous stage). Furthermore, at this stage two or more formulæ differing only in the name of quantified variables are reduced to one instance; and all inconsistent symbolic heaps that can be detected sintactically (denoted $\Pi \mid \Sigma \vdash \mathit{False}$ in fig. 3) are eliminated. As a consecuence, the number of chains that begin with a term $x \mapsto l, r, v$ (x a program variable) is bounded. In this way it is possible to prove:

Lemma 4. (Termination of abs)

1. *img*.abs *is finite.*
2. *The set of abstraction rules is strongly normalizing, i.e. every sequence of rewrites eventually terminates.*

The abstraction rules are not confluent, i.e. the obtained normal form after a terminating sequence of rewrites is not unique. The application order of the first stage rules is relevant as AbsArrow rules loose information that could be necessary to derive the application conditions of other rules. Our implementation obey the presented order. However, we do not observe significant differences when changing the application order in each stage.

The soundness of the rewriting rules of the previous section can be extended to include the abstraction rules:

Lemma 5. (Soundness of $\Longrightarrow$)

$$\text{If } \Pi \mid \Sigma \implies \Pi' \mid \Sigma', \text{ then } (s,h) \models \Pi \mid \Sigma \text{ implies } (s,h) \models \Pi' \mid \Sigma' \text{ for all } s, h$$

Previous results guarantee that the analysis given by the execution of the abstract semantics is sound and always terminates.

Theorem 6. *The abstract semantics is a sound over-approximation of the concrete semantics:* $[\![p]\!]_c.(\gamma.S) \subseteq \gamma.([\![p]\!]_a.S)$ *for all* $S \in \mathcal{P}(\mathsf{SH} \cup \{\top\})$. *Moreover, the algorithm defined by* $[\![\ ]\!]_a$ *is terminating.*

4 Managing Graphs

It is possible to use similar ideas to those underlying the predicate **trees** to describe general graph structures that includes cycles and sharing. The new predicate $\mathbf{graph}.\mathcal{C}.\mathcal{D}$ specifies with $\mathcal{C}$ the entry points of the graph, but unlike **trees** uses the multiset $\mathcal{D}$ to account for the dangling pointers that could point to nodes within the structure. Thus, these parameters resemble the ideas normally used to reason about graph algorithms: while $\mathcal{C}$ represents the entry points of paths to traverse, $\mathcal{D}$ realizes 'already visited' nodes. For a binary node graph, the formal semantics of **graph** is given by the least predicate satisfying:

$$\begin{aligned} \mathbf{graph}.\emptyset.\mathcal{D} &\doteq \mathbf{emp} \\ \mathbf{graph}.x \oplus \mathcal{C}.\mathcal{D} &\doteq ((x = \mathbf{nil} \vee x \in \mathcal{D}) \wedge \mathbf{graph}.\mathcal{C}.\mathcal{D}) \vee \\ &\quad ((x \neq \mathbf{nil} \wedge x \notin \mathcal{D}) \wedge (\exists l, r, v \cdot x \mapsto l, r, v * \mathbf{graph}.l \oplus r \oplus \mathcal{C}.x \oplus \mathcal{D})) \end{aligned}$$

To adapt our analysis to deal with graphs it is necessary to modify several rewrite rules that define it, while others will remain intact. In the rearr phase, rule Unfold must account for the differences in the definition of the predicate:

$$\frac{\Pi \mid \Sigma \vdash y = x \wedge y \neq \mathbf{nil} \wedge y \notin \mathcal{D}}{\Pi \mid \Sigma * \mathbf{graph}.y \oplus \mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * x \mapsto l', r', v' * \mathbf{graph}.l' \oplus r' \oplus \mathcal{C}.x \oplus \mathcal{D}} \text{[Unfold]}$$

Given the possible existence of cycles, the condition $y \notin \mathcal{D}$ could not always be derived. When rearr fail over a symbolic heap S because of this, it is replaced by the set of symbolic heaps obtained by adding the hypothesis $y \notin \mathcal{D} \vee y \in \mathcal{D}$. More precisely, S is replaced by the semantically equivalent set filter.$b.S$, where $b \doteq (y \neq e_1 \wedge \ldots \wedge y \neq e_n) \vee y = e_1 \vee \ldots \vee y = e_n$ given $\mathcal{D} = \{\!\!\{ e_1, \ldots, e_n \}\!\!\}$. Then rearr is relaunched on the new (richer) state.

The rules for the abs phase should take into account the possibility of cycles and sharing, plus the fact that a dangling pointer in $\mathcal{D}$ is not necessarily reachable from some entry point anymore. The most relevant rewriting rules are presented below:

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * \mathbf{graph}.y \oplus z \oplus \mathcal{C}.\mathcal{D} \qquad \Pi \mid \Sigma \vdash l = y \wedge r = z}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{graph}.x \oplus \mathcal{C}.x \ominus \mathcal{D}} \text{[AbsGraph1]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * \mathbf{graph}.y \oplus \mathcal{C}.\mathcal{D} \quad \Pi \mid \Sigma \vdash l = y \quad \Pi \mid \Sigma \vdash r = x \vee r = \mathbf{nil}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{graph}.x \oplus \mathcal{C}.x \ominus \mathcal{D}} \text{[AbsGraph2]}$$

$$\Pi \mid \Sigma * \mathbf{graph}.\mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{graph}.\mathbf{nil} \ominus \mathcal{C}.\mathbf{nil} \ominus \mathcal{D} \text{ [AbsGraph3]}$$

$$\frac{\Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma * \mathbf{graph}.x \oplus \mathcal{C}.y \oplus \mathcal{D} \implies \Pi \mid \Sigma * \mathbf{graph}.\mathcal{C}.y \oplus \mathcal{D}} \text{[AbsGraph4]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * y \mapsto \overline{e} * z \mapsto \overline{f} \qquad \Pi \mid \Sigma \vdash l = y \wedge r = z}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * x \mapsto l, r, v * \mathbf{graph}.\{\!\!\{ y, z \}\!\!\}.y \ominus z \ominus \{\!\!\{ \overline{e}, \overline{f} \}\!\!\}} \text{[AbsArrow1]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * y \mapsto \overline{e} \qquad \Pi \mid \Sigma \vdash l = y \quad \Pi \mid \Sigma \vdash r = x \vee r = \mathbf{nil}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * x \mapsto l, r, v * \mathbf{graph}.\{\!\!\{ y \}\!\!\}.y \ominus \{\!\!\{ \overline{e} \}\!\!\}} \text{[AbsArrow2]}$$

Table 1. Results of the shape analysis over several examples

Algorithm	Precondition / Postcondition	Link	Arrow	Inv.	Iter.	Time
`min/max`	$\mathbf{true} \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	2	4	2	2	0.003
	$x = \mathbf{nil} \mid \mathbf{emp}$ $x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$					
`destroy`	$\mathbf{true} \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	4	4	4	3	0.005
	$x = \mathbf{nil} \mid \mathbf{emp}$ $x \neq \mathbf{nil} \mid \mathbf{emp}$					
`search`	$\mathbf{true} \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	2	4	4	3	0.006
	$x = \mathbf{nil} \mid \mathbf{emp}$ $x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$					
`insert`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	3	3	12	4	0.036
	$\mathbf{true} \mid t \mapsto x', 0, 0 * x' \mapsto \mathbf{nil}, \mathbf{nil}, x''$ $\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$					
`toVine`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	3	4	8	6	0.061
	$\mathbf{true} \mid t \mapsto \mathbf{nil}, 0, 0$ $\mathbf{true} \mid t \mapsto x', 0, 0 * x' \mapsto \mathbf{nil}, \mathbf{nil}, x''$ $\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$					
`delete`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	3	2	14	5	0.142
	$\mathbf{true} \mid t \mapsto \mathbf{nil}, 0, 0$ $\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$					
`Schorr-Waite`	$r = x \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$	4	4	8	4	0.061
	$\mathbf{true} \mid \mathbf{emp}$ $x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\mid x \mid\!\}.\emptyset$ $r \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\mid r \mid\!\}.\emptyset$					
`Schorr-Waite`	$r = x \mid \mathbf{graph}.\{\!\mid x \mid\!\}.\emptyset$	4	4	17	4	0.185
	$x \neq \mathbf{nil} \mid \mathbf{graph}.\{\!\mid x \mid\!\}.\emptyset$ $r \neq \mathbf{nil} \mid \mathbf{graph}.\{\!\mid r \mid\!\}.\emptyset$ $\mathbf{true} \mid \mathbf{emp}$					

5 Experimental Results

We implemented our analysis in Haskell, and conducted experiments on an Intel Core Duo 1.86Ghz with 2GB RAM. In table 1 experimental results are presented for a variety of iterative algorithms on binary search trees adapted from GNU LibAVL [2], and an adaptation of Schorr-Waite traversal from [32].

Our implementation applies an abstraction phase on the final state to compact the postcondition, treating every variable not occurring in the precondition as a quantified one. Columns Link and Arrow represent the limits imposed over relevance levels to abstract a variable that forms a chain link, and a variable that occurs as an address of a "points-to" term, respectively. The column Inv. contains the number of states making up the invariant of the main cycle and the column Iter. the number of iterations needed to reach the fixed point. The execution time is measured in seconds. In all cases the used memory did not exceed the default allocation of 132 KB.

The `insert` and `delete` algorithms are the cases with a significant Arrow limit. This is because they present a deep pointer manipulation after the main loop, and therefore require a sufficiently precise invariant.

In the verification of the Schorr-Waite traversal over graphs, the computed invariant is really concise, consisting of seventeen symbolic heaps whose spatial part is some of the formulæ $\mathbf{graph}.\{\!|r, p|\!\}.\emptyset$, $\mathbf{graph}.\{\!|p|\!\}.\emptyset$, $\mathbf{graph}.\{\!|r|\!\}.\emptyset$, or **emp**, where r and p are the variables used to traverse de graph. This is a good example of the precision that our shape analysis is capable of achieving. Running the analysis applying the abstraction rules on quantified variables only, gives an invariant consisting of more than 180 formulæ.

We extended even more the analysis to deal with arrays of values, which enables the verification of Cheney's copying garbage collector algorithm from [30]. This extension is not presented here due to lack of space. The prototype implementation, code of examples and full results are available online [1].

6 Conclusions

Contributions. In this paper we introduced an abstract semantics that implicitly define an analysis able to automatically verify interesting properties about the shape of manipulated data structures, calculating loop invariants and postconditions. This semantics is an over-approximation of operational semantics on a standard memory model, hence, it could report false memory faults or leaks. Despite that, experiments show fitness between the computed abstract states and those expected according to the concrete semantics. It would be desirable to have a precise characterization of this relationship.

In order to define the analysis we have introduced novel linear predicates **trees** and **graph** to describe graph-like structures with multiple entry and outlet points. Good syntactic properties enable a simple caracterization of the abstraction phase. Preliminary experiments, based on variation of **tree** predicate and abstraction rules, are promising in verifying sorting and balancing invariants on BST trees and AVL trees.

The use of variable relevance level, although a very simple idea, introduces a significant improvement in the compactness of loop invariants, without impairing the necessary precision to obtain relevant postconditions. In some cases, a dramatic reduction in the number of formulæ making up an abstract state is achieved, making the entire process of analysis faster, and helping in the computation of intuitively understandable and correct invariants. Also it enables us to foresee a good behavior of the analysis on large-scale code. It would be interesting to extend the concept of Bi-Abduction of [7] to our domain to support incomplete specifications and procedure calls.

As a final consideration, by basing our analysis in [12] we inherit all advantages of Separation Logic. Our rewrite rules are valid implications whose semantic verification is quite simple, while symbolic execution rules derive directly from Hoare triples. Thus, our analysis is intuitive and its correctness easily verifiable.

Related Work. The main related works are [7,3,33] that derive from [12]. Proposed extensions implemented in SpaceInvader tool, aim at verifying real large-scale code, supporting only linear structures possibly combined in intrincate ways. Our work widen the domain of applicability of these methods by supporting non-linear structures. The predecessor work [4] introduces Smallfoot tool, based in a different kind of symbolic execution, although it supports binary trees. The tool reduces the verification of Hoare triples to logical implications between symbolic heaps but it does not compute loop invariants. The usual tree predicates seems adequated to verified recursive programs. In [25,15] the analysis presented are very similar to the ones in [12] both, in their technical aspects and their limitations.

Xisa tool [11] also uses Separation Logic to describe abstract states, but it uses generalized inductive predicates in the form of structure invariant checkers supplied by users. The fold and unfold of predicates guide automatic strategies for materialization (rearr in this work) and abstraction. Extension of [10] adds expressive power to specify certain relationships among summarized regions of heaps, as ordering invariants, back and cross pointers, etc, and a case study of item insertion in a red-black Tree is presented. The lack of examples impair the assessment of complexity this tool can handle.

Verifast tool [18] is also based in Separation Logic, allowing the verification of richly specified Hoare triples using inductive predicates for structures and pure recursive functions over those datatypes. It generates verification conditions discharged by a SMT Solver. No abstraction mechanism is provided, but loop invariants and instruction guiding fold/unfold of predicates must be manually annotated. It is able to verify a recursive implementation of binary trees library and the composite pattern (with its underlying graph structure) [19].

PALE system [24,20] allows the verification of programs specified with assertions in Weak Second-order Monadic Logic of Graph Types. Loops must be annotated with invariants and verification conditions are discharged in MONA tool [17]. The extension [14] enables a more efficient verification of algorithms on tree-shaped structures.

The analysis of [21] uses shape graphs and grammar annotations to specify non-cyclic data structures, discovering automatically descriptions for structures occurring at intermediate execution points. The analysis verify Schorr-Waite traversal and disposal on trees, and the construction of a binomial heap.

The parametric shape analysis of [28,5] based on 3-valued logic, together with their implementation TVLA [22], is the most general, powerful and used framework in the verification of shape properties on programs that involve complex manipulations on dynamic structures. This framework should be instantiated for each particular case through user defined intrumentation predicates that specify the type of supported structures and forms of abstraction. The instance presented in [23] allows the verification of partial correctness (as our analysis) and also termination of the Schorr-Waite tree traversal. Our proposal is less ambitious since our analysis is specialized in trees and graphs. However our abstract domains allow for high efficiency and precision in a wide class of algorithms, and the analysis support local reasoning which has shown great potential to scale on real large-scale programs as demonstrated in [7,33].

Acknowledgements. We are grateful to Dino Distefano for his encouragement to write this article, to Miguel Pagano for many discussions, and to the anonymous reviewers for their helpful comments for improving this work.

References

1. Details of experiments, http://cs.famaf.unc.edu.ar/~renato/seplogic.html
2. GNU LibAVL, http://www.stanford.edu/~blp/avl/
3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
5. Bogudlov, I., Lev-Ami, T., Reps, T.W., Sagiv, M.: Revamping TVLA: Making parametric shape analysis competitive. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 221–225. Springer, Heidelberg (2007)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 182–203. Springer, Heidelberg (2006)
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Shao, Z., Pierce, B.C. (eds.) ACM SIGPLAN-SIGACT 2009 Symposium on Principles of Programming Languages, pp. 289–300. ACM, New York (2009)
8. Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive resource invariant synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 259–274. Springer, Heidelberg (2009)
9. Calcagno, C., Parkinson, M.J., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007)
10. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: ACM SIGPLAN-SIGACT 2008 Symposium on Principles of Programming Languages, pp. 247–260. ACM, New York (2008)
11. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
12. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
13. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for java. In: Harris, G.E. (ed.) ACM SIGPLAN 2008 Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 213–226. ACM, New York (2008)
14. Elgaard, J., Møller, A., Schwartzbach, M.I.: Compile-time debugging of C programs working on trees. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 182–194. Springer, Heidelberg (2000)
15. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)

16. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: Ferrante, J., McKinley, K.S. (eds.) ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 266–277. ACM, New York (2007)
17. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
18. Jacobs, B., Piessens, F.: The verifast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (August. 2008)
19. Jacobs, B., Smans, J., Piessens, F.: Verifying the composite pattern using separation logic. In: SAVCBS Composite Pattern Challenge Track (2008)
20. Jensen, J.L., Jørgensen, M.E., Schwartzbach, M.I., Klarlund, N.: Automatic verification of pointer programs using monadic second-order logic. In: ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, pp. 226–236. ACM, New York (1997)
21. Lee, O., Yang, H., Yi, K.: Automatic verification of pointer programs using grammar-based shape analysis. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 124–140. Springer, Heidelberg (2005)
22. Lev-Ami, T., Sagiv, S.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)
23. Loginov, A., Reps, T.W., Sagiv, M.: Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)
24. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. ACM, New York (2001)
25. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
26. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
27. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
28. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
29. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
30. Torp-Smith, N., Birkedal, L., Reynolds, J.C.: Local reasoning about a copying garbage collector. ACM Trans. Program. Lang. Syst. 30(4) (2008)
31. Villard, J., Lozes, É., Calcagno, C.: Proving copyless message passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009)
32. Yang, H.: Local reasoning for stateful programs. PhD thesis, Champaign, IL, USA, Adviser-Uday S. Reddy (2001)
33. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

Modelling Metamorphism by Abstract Interpretation

Mila Dalla Preda[1], Roberto Giacobazzi[1], Saumya Debray[2], Kevin Coogan[2], and Gregg M. Townsend[2]

[1] Dipartimento di Informatica, Università di Verona
{mila.dallapreda,roberto.giacobazzi}@univr.it
[2] Department of Computer Science, University of Arizona
{debray,kpcoogan,gmt}@cs.arizona.edu

Abstract. Metamorphic malware apply semantics-preserving transformations to their own code in order to foil detection systems based on signature matching. In this paper we consider the problem of automatically extract metamorphic signatures from these malware. We introduce a semantics for self-modifying code, later called *phase semantics*, and prove its correctness by showing that it is an abstract interpretation of the standard trace semantics. Phase semantics precisely models the metamorphic code behavior by providing a set of traces of programs which correspond to the possible evolutions of the metamorphic code during execution. We show that metamorphic signatures can be automatically extracted by abstract interpretation of the phase semantics, and that regular metamorphism can be modelled as finite state automata abstraction of the phase semantics.

Keywords: Abstract interpretation, malware detection, metamorphic code, program transformation, static analysis, security, semantics.

1 Introduction

Challenges and insights. Detecting and neutralizing computer malware, such as worms, viruses, trojans, and spyware is a major challenge in modern computer security, involving both sophisticated intrusion detection strategies and advanced code manipulation tools and methods. Traditional misuse malware detectors (also known as *signature-based detectors*) are typically syntactic in nature: they use pattern matching to compare the byte sequence comprising the body of the malware against a *signature database* [22]. Malware writers have responded by using a variety of techniques in order to avoid detection: Encryption, oligomorphism with mutational decryptor patterns, and polymorphism with different encryption methods for generating an endless sequence of decryption patterns are typical strategies for achieving malware diversification. Metamorphism emerged in the last decade as an effective alternative strategy to foil detectors. Metamorphic malware apply semantics-preserving transformations to modify its own code so that one instance of the malware bears very little resemblance to another instance, in a kind of *body-polymorphism* [23], even though semantically, their functionality is the same. Thus, a metamorphic malware is a malware equipped with a *metamorphic engine* that takes the malware, or parts of it, as input and morphs it to a syntactically different but semantically equivalent variant in order to avoid detection. The quantity of metamorphic variants possible for a particular piece of malware makes it impractical to maintain a

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 218–235, 2010.
© Springer-Verlag Berlin Heidelberg 2010

signature set that is large enough to cover most or all of these variants, making standard signature-based detection ineffective [6]. Existing malware detectors therefore fall back on a variety of heuristic techniques, but these may be prone to false positives (where innocuous files are mistakenly identified as malware) or false negatives (where malware escape detection) at worst. The reason for this vulnerability to metamorphism lies upon the purely syntactic nature of most exiting and commercial detectors. The key for identifying metamorphic malware lies, instead, in a deeper understanding of their semantics. Still a major drawback of existing semantics-based methods (e.g., see [13, 19]) relies upon the *a priori* knowledge of the obfuscations used to implement the metamorphic engine. Because of this, it is always possible for any expert malware writer to develop alternative metamorphic strategies, even by simple modification of existing ones, able to foil any given detection scheme.

Contributions. We proposes a different approach to metamorphic malware detection based on the idea that *extracting metamorphic signatures is approximating malware semantics*. A *metamorphic signature* is therefore any (possibly decidable) approximation of the properties of code evolution. The semantics concerns here the way code changes, i.e., the effect of instructions that modify other instructions. We face the problem of determining how code mutates, yet catching properties of this mutation, without any a priori knowledge about the implementation of the metamorphic transformations. Traditional static analysis techniques are not adequate for this purpose, as they typically assume that programs do not change during execution. We therefore define a more general semantics-based behavioral model, called *phase semantics*, that can cope with changes to the program code at run time. The idea is to partition each possible execution trace of a metamorphic program into *phases*, each collecting the computations performed by a particular code variant. The sequence of phases (once disassembled) represents the sequence of possible code mutations, while the sequence of states within a given phase represents the behavior of a particular code variant. Abstract interpretation is then used to extract the invariant properties of phases, which are properties of the generated program variants. Abstract domains represent here properties of the code shape in phases. We use the domain of finite state automata (FSA) for approximating phases and provide a static semantics of traces of FSA as a computable abstraction of the phase semantics. We introduce the notion of *regular metamorphism* as a further approximation obtained by abstracting sequences of FSA into a single FSA. This abstraction provides an upper regular language-based approximation of *any* metamorphic behavior of a program. This is particularly suitable to extract metamorphic signatures for engines implemented themselves as FSA of basic code transformations, which correspond to the way most classical metamorphic generators are implemented [16, 20, 25]. Our approach is general and language independent, providing a systematic method for extracting approximate metamorphic signatures from any metamorphic malware P, in such a way that checking whether a given binary matches the metamorphic signature of P is decidable.

2 Background

Mathematical notation. Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \smallsetminus T$ the set-difference between S and T, with $S \subset T$ strict inclusion and

with $S \subseteq T$ inclusion. Let $S_\perp$ be set S augmented with the *undefined value* $\perp$, i.e., $S_\perp = S \cup \{\perp\}$. $\langle P, \leq \rangle$ denotes a poset P with ordering relation $\leq$, while a complete lattice P, with ordering $\leq$, least upper bound (lub) $\vee$, greatest lower bound (glb) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\perp$ is denoted by $\langle P, \leq, \vee, \wedge, \top, \perp \rangle$. $\sqsubseteq$ denotes pointwise ordering between functions. If $f : S \rightarrow T$ and $g : T \rightarrow Q$ then $g \circ f : S \rightarrow Q$ denotes the composition of f and g, i.e., $g \circ f = \lambda x.g(f(x))$. $f : P \rightarrow Q$ on posets is (Scott)-continuous when f preserves lub of countable chains in P. $f : C \rightarrow D$ on complete lattices is additive (co-additive) when for any $Y \subseteq C$, $f(\vee_C Y) = \vee_D f(Y)$ ($f(\wedge_C Y) = \wedge_D f(Y)$). Let A^* be the set of finite sequences, also called strings, of elements of A with ϵ the empty string, and with $|\omega|$ the length of string $\omega \in A^*$. We denote the concatenation of $\omega, \nu \in A^*$ as $\omega :: \nu$. We say that a string $s_0 \dots s_h$ is a subsequence of a string $t_0 \dots t_n$, denoted $s_0 \dots s_h \preceq t_0 t_1 \dots t_n$, if $\exists l \in [1, n] : \forall i \in [0, h] : \; s_i = t_{l+i}$.

Finite State Automata (FSA). An FSA M is a tuple (Q, δ, S, F, A), where Q is the set of states, $\delta : Q \times A \rightarrow \wp(Q)$ is the transition relation, $S \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states and A is the finite alphabet of symbols. Let $\omega \in A^*$, function $\delta^* : Q \times A^* \rightarrow \wp(Q)$ denotes the extension of δ to strings: $\delta^*(q, \epsilon) = \{q\}$ and $\delta^*(q, \omega s) = \bigcup_{q' \in \delta^*(q,\omega)} \delta(q', s)$. A string $\omega \in A^*$ is accepted by M if there exists $q_0 \in S : \; \delta^*(q_0, \omega) \cap F \neq \emptyset$. The language $\mathscr{L}(M)$ accepted by an FSA M is the set of all strings accepted by M. Given an FSA M and a partition π over its states, the *quotient automaton* $M/\pi = (Q', \delta', S', F', A)$ is defined as follows: $Q' = \{[q]_\pi \mid q \in Q\}$, $\delta' : Q' \times A \rightarrow \wp(Q')$ is the function $\delta'([q]_\pi, s) = \bigcup_{p \in [q]_\pi} \{[q']_\pi \mid q' \in \delta(p, s)\}$, $S' = \{[q]_\pi \mid q \in S\}$, and $F' = \{[q]_\pi \mid q \in F\}$. An FSA $M = (Q, \delta, S, F, A)$ can be equivalently specified as a graph $M = (Q, E, S, F)$ with a node $q \in Q$ for each automata state and a labeled edge $(q, s, q') \in E$ if and only if $q' \in \delta(q, s)$.

Abstract Interpretation. Abstract interpretation is based on the idea that the behaviour of a program at different levels of abstraction is an approximation of its (concrete) semantics [8, 9]. The concrete program semantics is computed on the concrete domain $\langle C, \leq_C \rangle$, while approximation is encoded by an abstract domain $\langle A, \leq_A \rangle$. In abstract interpretation abstraction is specified as a Galois connection (GC) (C, α, γ, A) , i.e., an adjunction [8,9], namely as an abstraction map $\alpha : C \rightarrow A$ and a concretization map $\gamma : A \rightarrow C$ such that: $\forall a \in A, c \in C : \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. Let A_1 and A_2 be abstract domains of the concrete domain C: A_1 is more precise than A_2 when $\gamma_2(A_2) \subseteq \gamma_1(A_1)$. Given a GC (C, α, γ, A) and a concrete predicate transformer (semantics) $F : C \rightarrow C$, we say that $F^\sharp : A \rightarrow A$ is a *sound* approximation of F in A if $\forall c \in C, \alpha(F(c)) \leq_A F^\sharp(\alpha(c))$. When $\alpha \circ F = F^\sharp \circ \alpha$, the abstract function $F^\sharp$ is a *complete* abstraction of F in A. While any abstract domain induces the canonical *best correct approximation* $\alpha \circ F \circ \gamma$ of $F : C \rightarrow C$ in A, not all abstract domains induce a complete abstraction [17]. The least fixpoint (lfp) of an operator F on a poset $\langle P, \leq \rangle$, when it exists, is denoted by $\mathit{lfp}^{\leq} F$, or by $\mathit{lfp} F$ when $\leq$ is clear. Any continuous operator $F : C \rightarrow C$ on a complete lattice $C = \langle C, \leq_C, \vee_C, \wedge_C, \top_C, \perp_C \rangle$ admits a lfp: $\mathit{lfp}^{\leq_C} F = \bigvee_{n \in \mathbb{N}} F^i(\perp_C)$, where for any $i \in \mathbb{N}$ and $x \in C$: $F^0(x) = x$; $F^{i+1}(x) = F(F^i(x))$. If $F^\sharp : A \rightarrow A$ is a correct approximation of $F : C \rightarrow C$ on $\langle A, \leq_A \rangle$, then $\alpha(\mathit{lfp}^{\leq_C} F) \leq_A \mathit{lfp}^{\leq_A} F^\sharp$. Convergence can be ensured through *widening*

Syntactic categories:

$n, a \in \mathbb{N}$	(naturals)
$e \in \mathbb{E}$	(expressions)
$I \in \mathbb{I}$	(instructions)
$\mathtt{m} \in \mathcal{M} : \mathbb{N} \to \mathbb{N}_\perp$	(memory map)
$P \in \mathcal{M} \times \mathbb{N} = \mathbb{P}$	(programs)

Expressions:

$e ::= n \mid \mathtt{MEM}[e] \mid \mathtt{MEM}[e_1]\ \mathtt{op}\ \mathtt{MEM}[e_2] \mid$
$\quad \mathtt{MEM}[e_1]\ \mathtt{op}\ n$

Instructions:

$I ::= \mathtt{call}\ e \mid \mathtt{ret} \mid \mathtt{pop}\ e \mid \mathtt{push}\ e \mid \mathtt{nop} \mid$
$\quad \mathtt{MEM}[e_1] := e_2 \mid \mathtt{input} \Rightarrow \mathtt{MEM}[e] \mid$
$\quad \mathtt{if}\ e_1\ \mathtt{goto}\ e_2 \mid \mathtt{goto}\ e \mid \mathtt{halt}$

Fig. 1. Syntax of an abstract assembly language

iterations along increasing chains [8]. A widening operator $\triangledown : P \times P \to P$ approximates the lub, i.e., $\forall X, Y \in P : X \leq_P (X \triangledown Y)$ and $Y \leq_P (X \triangledown Y)$, and it is such that the increasing chain W^i, where $W^0 = \bot$ and $W^{i+1} = W^i \triangledown F(W^i)$ is not strictly increasing for $\leq_P$. The limit of the sequence W^i provides an upper fixpoint approximation of F on P, i.e., $\mathit{lfp}^{\leq_P} F \leq_P \mathit{lim}_{i\to\infty} W^i$.

3 Modelling Metamorphism

Abstract assembly language. Executable programs make no fundamental distinction between code and data. This makes it possible to modify a program by operating on a memory location as though it contains data, e.g., by adding or subtracting some value from it, and then interpreting the result as code and executing it. To model this aspect, we define a program to be a pair $P = (\mathtt{m}, a)$, where $\mathtt{m}$ specifies the contents of a *memory* (both code and data) and a denotes the *entry point* of P, namely the address of the first instruction of P. Since a memory location contains a natural number that can be interpreted either as data or as instruction[1] we use an injective function $\mathtt{encode} : \mathbb{I} \to \mathbb{N}$ that, given an instruction $I \in \mathbb{I}$, returns its binary $\mathtt{encode}(I) \in \mathbb{N}$, and a function $\mathtt{decode} : \mathbb{N} \to \mathbb{I}_\perp$ that given a natural number n returns I if $\mathtt{encode}(I) = n$ otherwise $\perp$. Fig. 1 shows the syntax of our abstract assembly language, whose structure is inspired from real assembly languages. A program *state* is a tuple $\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle$ where $\mathtt{m}$ is the memory map, a is the address of the next instruction to be executed, $\theta \in \mathbb{N}^*$ is the stack and $\mathfrak{I} \in \mathbb{N}^*$ is the input string. Let $\Sigma = \mathbb{N}_\perp \times \mathcal{M} \times \mathbb{N}^* \times \mathbb{N}^*$ be the set of program states. The semantics of expressions is specified by a function $\mathcal{E} : \mathbb{E} \times \mathcal{M} \to \mathbb{N}$:

$$\mathcal{E}[\![n]\!]\mathtt{m} = n$$
$$\mathcal{E}[\![\mathtt{MEM}[e]]\!]\mathtt{m} = \mathtt{m}(\mathcal{E}[\![e]\!]\mathtt{m})$$
$$\mathcal{E}[\![\mathtt{MEM}[e_1]\ \mathtt{op}\ \mathtt{MEM}[e_2]]\!]\mathtt{m} = \mathcal{E}[\![\mathtt{MEM}[e_1]]\!]\mathtt{m}\ \mathtt{op}\ \mathcal{E}[\![\mathtt{MEM}[e_2]]\!]\mathtt{m}$$
$$\mathcal{E}[\![\mathtt{MEM}[e_1]\ \mathtt{op}\ n]\!]\mathtt{m} = \mathcal{E}[\![\mathtt{MEM}[e_1]]\!]\mathtt{m}\ \mathtt{op}\ n$$

and the semantics of instructions by a function $\mathcal{I} : \mathbb{I} \times \Sigma \to \Sigma$:

$$\mathcal{I}[\![\mathtt{call}\ e]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \langle \mathcal{E}[\![e]\!]\mathtt{m}, \mathtt{m}, (a+1) :: \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\mathtt{ret}]\!]\langle a, \mathtt{m}, n :: \theta, \mathfrak{I}\rangle = \langle n, \mathtt{m}, \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\mathtt{MEM}[e_1] := e_2]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \langle a+1, \mathtt{m}[\mathcal{E}[\![e_1]\!]\mathtt{m} \leftarrow \mathcal{E}[\![e_2]\!]\mathtt{m}], \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\mathtt{input} \Rightarrow \mathtt{MEM}[e]]\!]\langle a, \mathtt{m}, \theta, n :: \mathfrak{I}\rangle = \langle a+1, \mathtt{m}[\mathcal{E}[\![e]\!]\mathtt{m} \leftarrow n], \theta, \mathfrak{I}\rangle$$

[1] For simplicity, we assume that each instruction occupies a single location in memory, because the issues raised by variable-length instructions are orthogonal to the topic of this paper, and do not affect any of our results.

$$\mathcal{I}[\![\texttt{if } e_1 \texttt{ goto } e_2]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \begin{cases} \langle \mathcal{E}[\![e_2]\!]\mathtt{m}, \mathtt{m}, \theta, \mathfrak{I}\rangle & \text{if } \mathcal{E}[\![e_1]\!]\mathtt{m} \neq 0 \\ \langle a+1, \mathtt{m}, \theta, \mathfrak{I}\rangle & \text{otherwise} \end{cases}$$
$$\mathcal{I}[\![\texttt{pop } e]\!]\langle a, \mathtt{m}, n :: \theta, \mathfrak{I}\rangle = \langle a+1, \mathtt{m}[\mathcal{E}[\![e]\!]m \leftarrow n], \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\texttt{goto } e]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \langle \mathcal{E}[\![e]\!]\mathtt{m}, \mathtt{m}, \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\texttt{push } e]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \langle a+1, \mathtt{m}, \mathcal{E}[\![e]\!]m :: \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\texttt{halt}]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \langle \bot, \mathtt{m}, \theta, \mathfrak{I}\rangle$$
$$\mathcal{I}[\![\texttt{nop}]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle = \langle a+1, \mathtt{m}, \theta, \mathfrak{I}\rangle$$

Let $\mathcal{T} : \wp(\Sigma) \rightarrow \wp(\Sigma)$ be the *transition relation* between states, which is given by the point-wise extension of $\mathcal{T}(\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle) = \mathcal{I}[\![\texttt{decode}(\mathtt{m}(a))]\!]\langle a, \mathtt{m}, \theta, \mathfrak{I}\rangle$. As usual [11], the *maximal finite trace semantics* $\mathbf{S}[\![P]\!] \in \wp(\Sigma^*)$ of $P = (\mathtt{m}, a)$ is given by the lfp of $\mathcal{F}_\mathcal{T}[\![P]\!] : \wp(\Sigma^*) \rightarrow \wp(\Sigma^*)$ where $\mathit{Init}[\![P]\!] = \{\langle a, \mathtt{m}, \epsilon, \mathfrak{I}\rangle \mid \mathfrak{I} \text{ is an input stream}\}$ and $\mathcal{F}_\mathcal{T}[\![P]\!](X) = \mathit{Init}[\![P]\!] \cup \{\sigma\sigma_i\sigma_j \mid \sigma_j \in \mathcal{T}(\sigma_i), \sigma\sigma_i \in X\}$.

Phase Semantics. Intuitively, a *phase* is a maximal sequence of states in an execution trace that does not overwrite any memory location storing an instruction that is going to be executed later in the same trace. Given an execution trace $\sigma = \sigma_0 \ldots \sigma_n$, we can identify *phase boundaries* by considering the sets of memory locations modified by each state $\sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i\rangle$ with $i \in [0, n]$: *every time that a location a_j, with $i < j \leq n$, of a future instruction is modified by the execution of state σ_i, then the successive state σ_{i+1} is a phase boundary, since it stores a modified version of the code*. We consider the set $\mathit{mod}(\sigma_i) \subseteq \mathbb{N}$ of memory locations that are modified by the instruction executed in state σ_i:

$$\mathit{mod}(\sigma_i) = \begin{cases} \{\mathcal{E}[\![e]\!]\mathtt{m}\} & \text{if } \texttt{decode}(\mathtt{m}_i(a_i)) \in \{\texttt{MEM}[e] = e', \texttt{input} \Rightarrow \texttt{MEM}[e], \texttt{pop } e\} \\ \emptyset & \text{otherwise} \end{cases}$$

This allows us to formally define the phase boundaries and the phases of a trace.

Definition 1. *The set of phase boundaries of $\sigma = \sigma_0 \ldots \sigma_n \in \Sigma^*$, where $\forall i \in [0, n]$: $\sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i\rangle$, is: $\mathit{bound}(\sigma) = \{\sigma_0\} \cup \{\sigma_i \mid \mathit{mod}(\sigma_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}$. The set of phases of a trace $\sigma \in \Sigma^*$ is:*

$$\mathit{phases}(\sigma) = \left\{ \sigma_i \ldots \sigma_j \;\middle|\; \begin{array}{l} \sigma = \sigma_0 \ldots \sigma_i \ldots \sigma_j \sigma_{j+1} \ldots \sigma_n, \\ \sigma_i, \sigma_{j+1} \in \mathit{bound}(\sigma), \forall l \in [i+1, j] : \sigma_l \notin \mathit{bound}(\sigma) \end{array} \right\}$$

Observe that, by definition, the memory map of the first state of a phase always specifies the code snapshot that is executed in the same phase. Hence, the sequence of the initial states of the phases of a trace highlights the different code snapshots encountered during code execution. In general, different executions of a program give rise to different sequences of code snapshots. A complete characterization of all code snapshots of a self-modifying program can be obtained by organizing phases in a *program evolution graph*. Here, each vertex is a code snapshot P_i corresponding to a phase, and an edge $P_i \rightarrow P_j$ indicates that in some execution trace of the program, a phase with code snapshot P_i can be followed by a phase with code snapshot P_j.

Definition 2. *The program evolution graph of a program P_0 is $\mathbf{G}[\![P_0]\!] = (V, E)$:*

$$V = \{P_i = (\mathtt{m}_i, a_i) \mid \sigma = \sigma_0..\sigma_i..\sigma_n \in \mathbf{S}[\![P_0]\!] : \sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i \rangle \in \mathit{bound}(\sigma)\}$$

$$E = \left\{ (P_i, P_j) \,\middle|\, \begin{array}{l} P_i = (\mathtt{m}_i, a_i), P_j = (\mathtt{m}_j, a_j), \sigma = \sigma_0..\sigma_i..\sigma_{j-1}\sigma_j..\sigma_n \in \mathbf{S}[\![P_0]\!] : \\ \sigma_i = \langle a_i, \mathtt{m}_i, \theta_i, \mathfrak{I}_i \rangle, \sigma_j = \langle a_j, \mathtt{m}_j, \theta_j, \mathfrak{I}_j \rangle, \sigma_i \ldots \sigma_{j-1} \in \mathit{phases}(\sigma) \end{array} \right\}$$

A path in $\mathbf{G}[\![P_0]\!]$ is therefore a sequence of programs $P_0 \ldots P_n$ such that for every $i \in [0, n[$ we have that $(P_i, P_{i+1}) \in E$. Given a program P_0 the set of all possible (finite) paths of the program evolution graph $\mathbf{G}[\![P_0]\!]$ is the *phase semantics* of P_0, denoted $\mathbf{S}^{Ph}[\![P_0]\!]$: $\mathbf{S}^{Ph}[\![P_0]\!] = \{P_0 \ldots P_n \mid P_0 \ldots P_n \text{ is a path in } \mathbf{G}[\![P_0]\!]\}$.

P_0			
1:	$\mathtt{MEM}[f] := 100$	8:	$\mathtt{MEM}[\mathtt{MEM}[f]] := \mathtt{MEM}[4]$
2:	$\mathtt{input} \Rightarrow \mathtt{MEM}[a]$	9:	$\mathtt{MEM}[\mathtt{MEM}[f]+1] := \mathtt{MEM}[5]$
3:	$\mathtt{if}\ (\mathtt{MEM}[a] \bmod 2)\ \mathtt{goto}\ 7$	10:	$\mathtt{MEM}[\mathtt{MEM}[f]+2] := \mathtt{encode}(\mathtt{goto}\ 6)$
4:	$\mathtt{MEM}[b] := \mathtt{MEM}[a]$	11:	$\mathtt{MEM}[4] := \mathtt{encode}(\mathtt{nop})$
5:	$\mathtt{MEM}[a] := \mathtt{MEM}[a]/2$	12:	$\mathtt{MEM}[5] := \mathtt{encode}(\mathtt{goto}\ \mathtt{MEM}[f])$
6:	$\mathtt{goto}\ 8$	13:	$\mathtt{MEM}[f] := \mathtt{MEM}[f]+3$
7:	$\mathtt{MEM}[a] := (\mathtt{MEM}[a]+1)/2$	14:	$\mathtt{goto}\ 2$

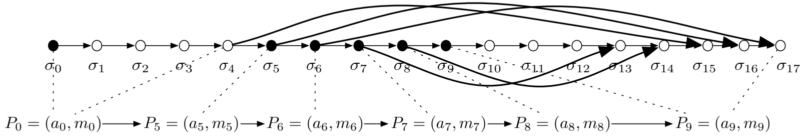


Fig. 2. A metamorphic program P_0 and the phases of one of its traces

Consider for instance the metamorphic program P_0 of Fig. 2. The metamorphic engine of P_0, which is stored at memory locations from 8 to 13, writes a nop at memory location 4 and copies the original content of this location to the free location identified by $\mathtt{MEM}[f]$; then it adds some goto instructions to preserve the original semantics. We consider the execution trace $\sigma = \sigma_0\sigma_1 \ldots \sigma_{17}$ of program P_0 corresponding to the input sequence $\mathfrak{I} = 7 :: 6$, in particular $\sigma = \langle 1, \mathtt{m}_0, \epsilon, 7 :: 6\rangle\langle 2, \mathtt{m}_1 = \mathtt{m}_0[f \leftarrow 100], \epsilon, 7 :: 6\rangle\langle 3, \mathtt{m}_2 = \mathtt{m}_1[a \leftarrow 7], \epsilon, 6\rangle\langle 7, \mathtt{m}_3 = \mathtt{m}_2, \epsilon, 6\rangle\langle 8, \mathtt{m}_4 = \mathtt{m}_3[a \leftarrow 4], \epsilon, 6\rangle\langle 9, \mathtt{m}_5 = \mathtt{m}_4[100 \leftarrow \mathtt{encode}(\mathtt{MEM}[b] := \mathtt{MEM}[a])], \epsilon, 6\rangle \ldots \langle 17, \mathtt{m}_{17} = \mathtt{m}_{16}[a \leftarrow 3], \epsilon, \epsilon\rangle$. Fig. 2 shows the considered execution trace σ where: the bold arrows denote the modifications of instructions that will be later executed, for example the bold arrow from $\sigma_4 = \langle a_4, \mathtt{m}_4, \theta_4, \mathfrak{I}_4\rangle$ to $\sigma_{15} = \langle a_{15}, \mathtt{m}_{15}, \theta_{15}, \mathfrak{I}_{15}\rangle$ means that location a_{15} is overwritten by the execution of instruction $\mathtt{decode}(\mathtt{m}_4(a_4))$ at state σ_4, i.e., $a_{15} \in \mathit{mod}(\sigma_4)$; and the black dots identify the states that are phase boundaries.

Fixpoint phase semantics. We introduce the notion of *mutating transition*, i.e., a transition between two states that leads to a state which is a phase boundary. We say that a pair of states (σ_i, σ_j) is a mutating transition of P_0, denoted $(\sigma_i, \sigma_j) \in \mathtt{MT}(P_0)$, if there exists a trace $\sigma = \sigma_0 \ldots \sigma_i\sigma_j \ldots \sigma_n \in \mathbf{S}[\![P_0]\!]$ such that $\sigma_j \in \mathit{bound}(\sigma)$. This allows

us to define the code transformer $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ that associates with each set of programs the set of their possible metamorphic variants: $P_j \in \mathcal{T}^{Ph}(P_i)$ means that during execution program P_i can be transformed into program P_j.

Definition 3. $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ *is given by the point-wise extension of:*

$$\mathcal{T}^{Ph}(P_0) = \left\{ P_l \;\middle|\; \begin{array}{l} P_l = (m_l, a_l), \sigma = \sigma_0 \ldots \sigma_{l-1}\sigma_l \in \mathbf{S}[\![P_0]\!], \sigma_l = \langle a_l, \mathfrak{m}_l, \theta_l, \mathfrak{I}_l \rangle, \\ (\sigma_{l-1}, \sigma_l) \in MT(P_0), \forall i \in [0, l-1[: (\sigma_i, \sigma_{i+1}) \notin MT(P_0) \end{array} \right\}$$

$\mathcal{T}^{Ph}$ can be extended to traces $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!] : \wp(\mathbb{P}^*) \to \wp(\mathbb{P}^*)$ as: $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!](Z) = P_0 \cup \{zP_iP_j \mid P_j \in \mathcal{T}^{Ph}(P_i), zP_i \in Z\}$.

Theorem 1. $lfp^{\subseteq}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!] = \mathbf{S}^{Ph}[\![P_0]\!]$.

A program Q is a metamorphic variant of a program P_0, denoted $P_0 \rightsquigarrow_{Ph} Q$, if Q is an element of at least one sequence in $\mathbf{S}^{Ph}[\![P_0]\!]$.

Correctness and completeness of phase semantics. We prove the correctness of phase semantics by showing that it is a sound approximation of trace semantics, namely by providing a pair of adjoint maps $\alpha_{Ph} : \wp(\Sigma^*) \to \wp(\mathbb{P}^*)$ and $\gamma_{Ph} : \wp(\mathbb{P}^*) \to \wp(\Sigma^*)$, for which the fixpoint computation of $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$ approximates the fixpoint computation of $\mathcal{F}_{\mathcal{T}}[\![P_0]\!]$. Given $\sigma = \langle a_0, \mathfrak{m}_0, \theta_0, \mathfrak{I}_0 \rangle \ldots \sigma_{i-1}\sigma_i \ldots \sigma_n$ we define α_{Ph} as:

$$\alpha_{Ph}(\sigma) = (\mathfrak{m}_0, a_0)\alpha_{Ph}(\sigma_i \ldots \sigma_n) \text{ s.t. } \sigma_i \in bound(\sigma), \forall l \in [0, i-1] : \sigma_l \notin bound(\sigma)$$

Abstraction α_{Ph} observes only the states of a trace that are phase boundaries and it can be lifted point-wise to $\wp(\Sigma^*)$ giving rise to the GC $(\wp(\Sigma^*), \alpha_{Ph}, \gamma_{Ph}, \wp(\mathbb{P}^*))$. The following result shows the correctness of the phase semantics.

Theorem 2. $\forall X \in \wp(\Sigma^*) : \alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}}[\![P_0]\!](X)) \subseteq \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!](\alpha_{Ph}(X))$.

The converse may not hold: $\alpha_{Ph}(X \cup \mathcal{F}_{\mathcal{T}}[\![P_0]\!](X)) \subset \alpha_{Ph}(X) \cup \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!](\alpha_{Ph}(X))$. In fact, given $X \in \wp(\Sigma^*)$, the concrete function $\mathcal{F}_{\mathcal{T}}[\![P_0]\!]$ makes only one transition in $\mathcal{T}$ and this may not be a mutating transition, while the abstract function $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$ *jumps* directly to the next mutating transition. Even if the fixpoint of $\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$ is not step-wise complete, it is complete at the fixpoint, as shown by the following theorem.

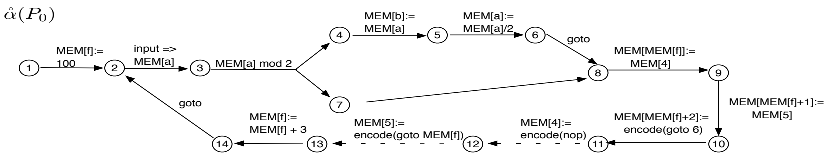
Theorem 3. $\alpha_{Ph}(lfp^{\subseteq}\mathcal{F}_{\mathcal{T}}[\![P_0]\!]) = lfp^{\subseteq}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]$.

4 Abstracting Metamorphism

Our model of metamorphic code behaviour is based on a very low-level representation of programs as memory maps that simply give the contents of memory locations together with the address of the instruction to be executed next. While such a representation is necessary to precisely capture the effects of code self-modification, it is not a convenient representation if we want to statically analyze the different code snapshots encountered during a program's execution. Our idea is to design an abstract interpretation of phase semantics, namely to approximate the computation of phase semantics on an abstract domain that captures properties of the evolution of the code, rather than

of the evolution of program states, as usual in abstract interpretation. We have to: (1) Define an abstract domain $\langle A, \sqsubseteq_A \rangle$ of *code properties* such that $(\wp(\mathbb{P}^*), \alpha_A, \gamma_A, A)$; (2) Define the abstract transition $\mathcal{T}^A : \wp(A) \to \wp(A)$ and $\mathcal{F}_{\mathcal{T}^A}[\![P_0]\!] : A \to A$ such that $lfp^{\sqsubseteq_A} \mathcal{F}_{\mathcal{T}^A}[\![P_0]\!] = \mathbf{S}^A[\![P_0]\!]$; (3) Prove that $\mathbf{S}^A[\![P_0]\!]$ is a correct approximation of phase semantics $\mathbf{S}^{Ph}[\![P_0]\!]$, i.e., $\alpha_A(lfp^{\subseteq} \mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \sqsubseteq_A \mathbf{S}^A[\![P_0]\!]$. This proves that $\mathbf{S}^A[\![P_0]\!]$ is such that *a program Q is a metamorphic variant of program P_0 with respect to A, denoted $P_0 \rightsquigarrow_A Q$, if $\mathbf{S}^A[\![P_0]\!]$ approximates Q in the abstract domain A*: $P_0 \rightsquigarrow_A Q \quad \Leftrightarrow \quad \alpha_A(Q) \sqsubseteq_A \mathbf{S}^A[\![P_0]\!]$. In this sense, $\mathbf{S}^A[\![P_0]\!]$ is an *abstract metamorphic signature* for P_0. Abstract domains for code properties need to approximate properties of sequences of instructions. This can be achieved naturally by grammar-based, constraint-based and finite state automata abstractions. In the following we propose to abstract programs by a FSA describing the sequence of (possibly abstract) instructions that may be disassembled from the given memory.

Phases as FSA. The most commonly used program representation is the *control flow graph*. In this representation, the vertices contain the instructions to be executed, and the edges represent possible control flow. For our purposes, it is convenient to consider a dual representation where vertices correspond to program locations and abstract instructions label edges. Let M_P denote the FSA-representation of a given program P and let $\mathscr{L}(M_P)$ be the language it recognizes. The idea is that for each sequence in $\mathscr{L}(M_P)$ the order of the instructions in the sequence corresponds to the execution order of the corresponding concrete instructions in at least one run of the control flow graph of P. Instructions are abstracted in order to provide a simplified alphabet. In the rest of the paper, for the sake of simplicity, we consider function $\iota : \mathbb{I} \to \mathring{\mathbb{I}}$ defined in Fig. 3. Let $\rho : \mathbb{I} \times \mathbb{N} \to \wp(\mathbb{N})$ denote any sound control flow analysis that determines the possible successors of a given instruction at a given location, namely



$$\iota(I) = \mathring{I} = \begin{cases} \texttt{call} & \text{if } I = \texttt{call } e \\ e_1 & \text{if } I = \texttt{if } e_1 \texttt{ goto } e_2 \\ \texttt{goto} & \text{if } I = \texttt{goto } e \\ I & \text{otherwise} \end{cases}$$

Edges$(P = (\mathtt{m}, a), Q_P, \rho)$
$E_P = \emptyset$
while $Q_P \neq \emptyset$
  select $b \in Q_P$ and $Q_P = Q_P \smallsetminus \{b\}$
  $I = \mathtt{decode}(\mathtt{m}(b))$
  for each $c \in \rho(I, b) \cap Q_P$
    $E_P = E_P \cup \{(b, \iota(I), c)\}$
return E_P

Fig. 3. FSA $\mathring{\alpha}(P_0)$ corresponding to program P_0 of Fig. 2, instruction abstraction $\iota : \mathbb{I} \to \mathring{\mathbb{I}}$ and the algorithm that computes E_P

$\rho(I, b)$ associates with instruction I stored at memory location b the set of locations of its possible successors. Let $\mathfrak{F}$ be the set of FSA over the alphabet $\mathring{\mathbb{I}}$ of abstract instructions where every state is considered to be final. Each FSA in $\mathfrak{F}$ is specified as a graph $M = (Q, E, S)$. We define function $\mathring{\alpha} : \mathbb{P} \rightarrow \mathfrak{F}$ that associates with each program $P = (\mathtt{m}, a)$ its corresponding FSA-representation as follows: $\mathring{\alpha}(P) = (Q_P, E_P, \{a\})$ where $Q_P = \{b \mid \mathtt{decode}(\mathtt{m}(b)) \in \mathbb{I}\}$ is the set of locations that store an instruction of P, and the set of edges $E_P \subseteq Q_P \times \mathring{\mathbb{I}} \times Q_P$ is computed by the algorithm **Edges** in Fig. 3. This algorithm, given $P = (m, a)$, starts by initializing E_P to the empty set and then for every memory location b that stores an instruction I it adds an edge labeled with $\iota(I)$, whose source is the location b and whose destinations are the locations in $\rho(I, b)$. As an example, at the top of Fig. 3 we show the automaton $\mathring{\alpha}(P_0)$ corresponding to program P_0 of Fig. 2. We say that $\pi = a_0[\mathring{I}_0] \ldots [\mathring{I}_{n-1}]a_n[\mathring{I}_n]a_{n+1}$ is a *path* of automaton $M = (Q, E, S)$, denoted $\pi \in \Pi(M)$, if $a_0 \in S$ and $\forall i \in [0, n[$: $(a_i, \mathring{I}_i, a_{i+1}) \in E$. Observe that even if the alphabet $\mathring{\mathbb{I}}$ is unbounded (due to the unlimited number of possible expressions), the FSA-representation of every program uses only a finite subset of alphabet $\mathring{\mathbb{I}}$. By point-wise extension of function $\mathring{\alpha}$ we obtain the GC $(\wp(\mathbb{P}), \mathring{\alpha}, \mathring{\gamma}, \wp(\mathfrak{F}))$. Note that abstraction ι defined above makes the FSA-representation of programs independent (up to renaming) from program position.

Theorem 4. *If P_1 and P_2 differ only in their memory position then $\mathring{\alpha}(P_1)$ and $\mathring{\alpha}(P_2)$ are equivalent up to address renaming.*

Abstract phase semantics as traces of FSA. Let $\alpha_{\mathfrak{F}} : \mathbb{P}^* \rightarrow \mathfrak{F}^*$ be the extension of $\mathring{\alpha} : \mathbb{P} \rightarrow \mathfrak{F}$ to sequences: $\alpha_{\mathfrak{F}}(\epsilon) = \epsilon$ and $\alpha_{\mathfrak{F}}(P_0 P_1 \ldots P_n) = \mathring{\alpha}(P_0)\alpha_{\mathfrak{F}}(P_1 \ldots P_n)$. $\alpha_{\mathfrak{F}}$ can be lifted point-wise to $\wp(\mathbb{P}^*)$ and it gives rise to the GC $(\wp(\mathbb{P}^*), \alpha_{\mathfrak{F}}, \gamma_{\mathfrak{F}}, \wp(\mathfrak{F}^*))$. In order to compute a correct approximation of the phase semantics on $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$, we need to define an abstract transition relation $\mathcal{T}^{\mathfrak{F}} : \wp(\mathfrak{F}) \rightarrow \wp(\mathfrak{F})$ on FSA that correctly approximates $\mathcal{T}^{Ph} : \wp(\mathbb{P}) \rightarrow \wp(\mathbb{P})$. One possibility is to define $\mathcal{T}^{\mathfrak{F}}$ as the best correct approximation of $\mathcal{T}^{Ph}$ on $\wp(\mathfrak{F})$, namely $\mathcal{T}^{\mathfrak{F}} = \mathring{\alpha} \circ \mathcal{T}^{Ph} \circ \mathring{\gamma}$, and function $\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!] : \wp(\mathfrak{F}^*) \rightarrow \wp(\mathfrak{F}^*)$ as follows: $\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!](K) = \mathring{\alpha}(P_0) \cup \{kM_iM_j \mid kM_i \in K, M_j \in \mathcal{T}^{\mathfrak{F}}(M_i)\}$. From $\mathcal{T}^{\mathfrak{F}}$ correctness we have $\mathbf{S}^{\mathfrak{F}}[\![P_0]\!] = \mathit{lfp}\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!]$ correctness.

Theorem 5. $\alpha_{\mathfrak{F}}(\mathit{lfp}\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \subseteq \mathit{lfp}\mathcal{F}_{\mathcal{T}^{\mathfrak{F}}}[\![P_0]\!] = \mathbf{S}^{\mathfrak{F}}[\![P_0]\!]$.

$\mathbf{S}^{\mathfrak{F}}[\![P_0]\!]$ approximates phase semantics by abstracting programs with FSA, while the transitions, i.e., the effect of the metamorphic engine, follow directly from $\mathcal{T}^{Ph}$ and are not approximated. For this reason $\mathbf{S}^{\mathfrak{F}}[\![P_0]\!]$ is not computable in general. In the following we introduce a static computable approximation of the transition relation on FSA that allows us to obtain a static approximation $\mathbf{S}^{\sharp}[\![P_0]\!]$ of the phase semantics of P_0 on $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$. $\mathbf{S}^{\sharp}[\![P_0]\!]$ may play the role of abstract metamorphic signature of P_0. To this end, we introduce the notion of *limits* of a path that approximates the notion of bounds of a trace, and the notion of *transition edge* that approximates the notion of mutating transition. Moreover, we assume to have access to the following sound program analyses for P_0:

– a stack analysis $\mathit{StackVal} : \mathbb{N} \rightarrow \wp(\mathbb{N})$ that approximates the set of possible values on the top of the stack when control reaches a given location (e.g. [1,2]);

– a memory analysis $LocVal : \mathbb{N} \times \mathbb{N} \rightarrow \wp(\mathbb{N})$ that approximates the set of possible values that can be stored in a memory location when the control reaches a given location (e.g. [1,2]).

These analyses allow us to define $EVal : \mathbb{N} \times \mathbb{E} \rightarrow \wp(\mathbb{N})$, that approximates the evaluation of an expression in a given point:

$EVal(b, n) = \{n\}$
$EVal(b, \texttt{MEM}[e]) = \{LocVal(b, l) \mid l \in EVal(b, e)\}$
$EVal(b, \texttt{MEM}[e_1] \texttt{ op } \texttt{MEM}[e_2]) = \{n_1 \texttt{ op } n_2 \mid i \in \{1, 2\} : n_i \in EVal(b, \texttt{MEM}[e_i])\}$
$EVal(\texttt{MEM}[e] \texttt{ op } n) = \{n_1 \texttt{ op } n \mid n_1 \in EVal(b, \texttt{MEM}[e])\}$

and a sound control flow analysis $\rho : \mathbb{I} \times \mathbb{N} \rightarrow \wp(\mathbb{N})$:

$\rho(\texttt{call } e, b) = \rho(\texttt{goto } e) = EVal(b, e)$
$\rho(\texttt{ret}, b) = StackVal(b)$
$\rho(\texttt{if } e_1 \texttt{ goto } e_2, b) = \{b + 1\} \cup EVal(b, e_2)$
$\rho(\texttt{halt}, b) = \emptyset$
$\rho(I, b) = \{b + 1\}$ in all other cases

Moreover, we define $write : \mathring{\mathbb{I}} \times \mathbb{N} \rightarrow \wp(\mathbb{N})$ approximating the set of locations that may be modified by the execution of an abstract instruction memorized at a given location:

$$write(\mathring{I}, b) = \begin{cases} EVal(b, e_1) & \text{if } \mathring{I} = \texttt{MEM}[e_1] := e_2 \\ EVal(b, e) & \text{if } \mathring{I} \in \{\texttt{input} \Rightarrow \texttt{MEM}[e], \texttt{pop } e\} \\ \emptyset & \text{otherwise} \end{cases}$$

We define the *limits* of a path π as the nodes that are reached by an edge labeled by an abstract instruction that may modify the label of a future edge in π, namely an abstract instruction that occurs later in the same path. Given a path $\pi = a_0[\mathring{I}_0] \dots [\mathring{I}_{n-1}]a_n$ we have: $limit(\pi) = \{a_0\} \cup \{a_i \mid write(\mathring{I}_{i-1}, a_{i-1}) \cap \{a_j \mid i \leq j \leq n\} \neq \emptyset\}$.

Definition 4. *A pair of program locations (b, c) is a transition edge of $M = (Q, E, S)$, denoted $(b, c) \in \texttt{TE}(M)$, if there exists $a \in S$: $\pi = a[\mathring{I}_a] \dots [\mathring{I}_{b-1}]b[\mathring{I}_b]c \in \Pi(M)$ and $c \in limit(\pi)$.*

In the FSA of Fig. 3 the transition edges are the dashed ones since the instructions labeling these edges overwrite a location that is reachable in the future. Observe that also the instructions labeling the edges from 8 to 9, from 9 to 10, and from 10 to 11 write instructions in memory, but the locations that store these instructions are not reachable when considering the control flow of P_0.

In order to statically compute the set of possible FSA evolution of a given automaton $M = (Q, E, S)$ we need to statically execute the abstract instructions that may modify an FSA. Algorithm $\mathbf{EXE}(M, \mathring{I}, b)$ in Fig. 4 returns the set Exe of all possible FSA that can be obtained by executing instruction $\mathring{I}$ stored at location b of automaton M. The algorithm starts by initializing Exe to the FSA M' that has the same states and edges of M and whose possible initial states S' are the nodes reachable through instruction $\mathring{I}$ stored at b in M. This ensures correctness when the execution of instruction $\mathring{I}$ does not correspond to a real code mutation. Then if $\mathring{I}$ writes in memory we consider the set X of locations that it can modify and the set Y of possible instructions that it can write,

$\mathbf{EXE}(M, \mathring{I}, b)$ // $M = (Q, E, S)$ *is a FSA*
$Exe = \{M' = (Q, E, S') \mid S' = \{d \mid (b, \mathring{I}, d) \in E\}\}$
if $\mathring{I} = \texttt{MEM}[e_1] := e_2$
  then $X = write(\mathring{I}, b)$
  $Y = \{n \mid n \in EVal(b, e_2), \texttt{decode}(n) \in \mathbb{I}\}$
  $Exe = Exe \cup \mathbf{NEXT}(X, Y, M, b)$
if $\mathring{I} = \texttt{input} \Rightarrow \texttt{MEM}[e]$
  then $X = write(\mathring{I}, b)$
  $Y = \{n \mid n$ is an input $, \texttt{decode}(n) \in \mathbb{I}\}$
  $Exe = Exe \cup \mathbf{NEXT}(X, Y, M, b)$
if $\mathring{I} = \texttt{pop}\ \ e$
  then $X = write(\mathring{I}, b)$
  $Y = \{n \mid n \in StackVal(b), \texttt{decode}(n) \in \mathbb{I}\}$
  $Exe = Exe \cup \mathbf{NEXT}(X, Y, M, b)$
return Exe

$\mathbf{NEXT}(X, Y, M, b)$
$Next = \emptyset$
while $X \neq \emptyset$
  select a_j from X and $X = X \smallsetminus \{a_j\}$
  $\hat{E} = E \smallsetminus \{(a_j, \mathring{I}_j, c) \mid (a_j, \mathring{I}_j, c) \in E\}$
  $Next = Next \cup \bigcup_{n \in Y}\{\ \hat{M} = (\hat{Q}, \hat{E}, \hat{S}) \mid$
    $\hat{Q} = Q \cup \{a_j\} \cup \rho(\texttt{decode}(n), a_j)$
    $\hat{E} = \hat{E} \cup \{(a_j, \iota(\texttt{decode}(n)), d) \mid$
      $d \in \rho(\texttt{decode}(n), a_j)\}$
    $\hat{S} = \{d \mid (b, \mathring{I}, d) \in E\}\ \}$
return $Next$

Fig. 4. Algorithm for statically executing instruction $\mathring{I}$

and we add to Exe the set of all possible automata that can be obtained by writing an instruction of Y in a memory location in X, i.e., $\mathbf{NEXT}(X, Y, M, b)$.

Let $Succ(M)$ denote the possible evolutions of automaton M, namely the automata that can be obtained by the execution of the abstract instruction labeling the first transition edge of a path of M:

$$Succ(M) = \left\{ M' \;\middle|\; \begin{array}{l} a_0[\mathring{I}_0] \ldots [\mathring{I}_{l-1}]a_l[\mathring{I}_l]a_{l+1} \in \Pi(M), (a_l, a_{l+1}) \in \texttt{TE}(M), \\ \forall i \in [0, l[\colon (a_i, a_{i+1}) \notin \texttt{TE}(M), M' \in \mathbf{EXE}(M, \mathring{I}_l, a_l) \end{array} \right\}$$

We can now define the static transition $\mathcal{T}^\sharp : \wp(\mathfrak{F}) \rightarrow \wp(\mathfrak{F})$. The idea is that the possible static successors of an automaton M are all the automata in $Succ(M)$ together with all the automata M' that are different from M and that can be reached from M through a sequence of successive automata that differ from M only in the entry point. This ensures the correctness of $\mathcal{T}^\sharp$, i.e., $M_l \in \mathcal{T}^{\mathfrak{F}}(M_0) \Rightarrow M_l \in \mathcal{T}^\sharp(M_0)$, even if between M_0 and M_l there are transition edges that do not correspond to any mutating transition.

Definition 5. *Let* $M = (Q, E, S)$. $\mathcal{T}^\sharp : \wp(\mathfrak{F}) \rightarrow \wp(\mathfrak{F})$ *is given by the point-wise extension of:*

$$\mathcal{T}^\sharp(M) = Succ(M) \cup \left\{ M' \;\middle|\; \begin{array}{l} MM_1 \ldots M_k M' : M_1 \in Succ(M), \forall i \in [1, k[\colon \\ M_{i+1} \in Succ(M_i), M' = (Q', E', S') \in Succ(M_k), \\ (E \neq E' \vee Q \neq Q'), \forall j \in [1, k] : M_j = (Q, E, S_j) \end{array} \right\}$$

This allows us to define function $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!] : \wp(\mathfrak{F}^*) \rightarrow \wp(\mathfrak{F}^*)$ that statically approximates the iterative computation of phase semantics on the abstract domain $\langle \wp(\mathfrak{F}^*), \subseteq \rangle$ as follows: $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!](K) = \mathring{\alpha}(P_0) \cup \{kM_iM_j \mid (M_i, M_j) \in \mathcal{T}^\sharp, kM_i \in K\}$. The following result shows the correctness of $\mathbf{S}^\sharp[\![P_0]\!] = lfp\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$.

Theorem 6. $\alpha_{\mathfrak{F}}(lfp\mathcal{F}_{\mathcal{T}^{Ph}}[\![P_0]\!]) \subseteq lfp\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$.

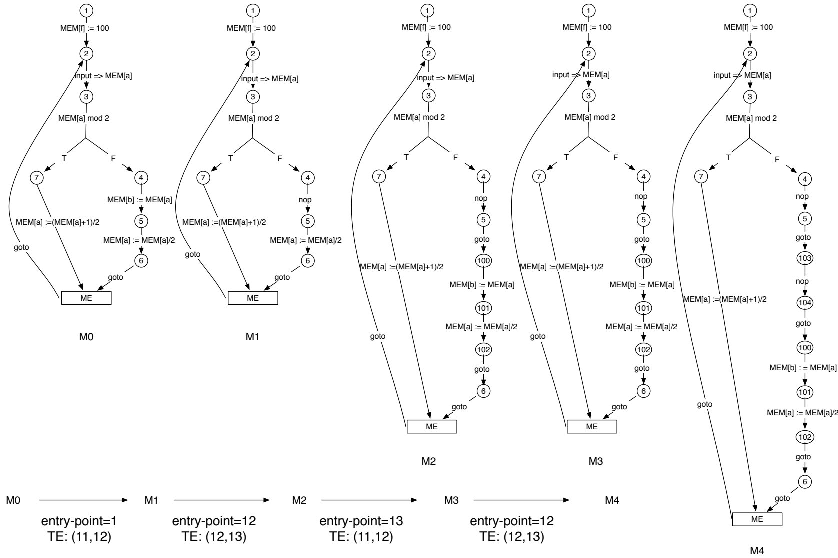


Fig. 5. Some metamorphic variants of program P_0 of Fig. 2, where the metamorphic engine, namely the instructions stored at locations from 8 to 14, is briefly represented by the box marked ME. In the graphic representation of automata we omit to show the nodes that are not reachable.

In Fig. 5 we report a possible sequence of FSA that can be generated during the execution of program P_0 of Fig. 2. In this case, thanks to the simplicity of the example, it is possible to use the transition relation over FSA defined by $\mathcal{T}^{\mathfrak{F}}$.

5 Widening Phases for Regular Metamorphism

Regular metamorphism models the metamorphic behaviour as a regular language of abstract instructions. This can be achieved by approximating sequences of FSA into a single FSA, denoted $\mathbf{W}[\![P_0]\!]$. $\mathbf{W}[\![P_0]\!]$ represents all possible (regular) program evolutions of P_0, i.e., it recognizes all the sequences of instructions that correspond to a run of at least one metamorphic variant of P_0. This abstraction is able to precisely model metamorphic engines implemented as FSA of basic code replacement as well as it may provide a regular language-based approximation for any metamorphic engine, by extracting the *regular* invariant of their behaviour.

It is known that FSA can be ordered according to the language they recognize: $M_1 \sqsubseteq_{\mathfrak{F}} M_2$ if $\mathscr{L}(M_1) \subseteq \mathscr{L}(M_2)$. Observe that $\sqsubseteq_{\mathfrak{F}}$ is reflexive and transitive but not antisymmetric and it is therefore a pre-order. Moreover, according to this ordering, an unique least upper bound of two automata M_1 and M_2 does not always exist, since there is an infinite number of automata that recognize the language $\mathscr{L}(M_1) \cup \mathscr{L}(M_2)$.

Given two automata $M_1 = (Q_1, \delta_1, S_1, F_1, A_1)$ and $M_2 = (Q_2, \delta_2, S_2, F_2, A_2)$, we approximate their least upper bound as follows:

$$M_1 \Cup M_2 = (Q_1 \cup Q_2, \hat{\delta}, S_1 \cup S_2, F_1 \cup F_2, A_1 \cup A_2)$$

where $\hat{\delta} : (Q_1 \cup Q_2) \times (A_1 \cup A_2) \rightarrow \wp(Q_1 \cup Q_2)$ is defined as $\hat{\delta}(q, s) = \delta_1(q, s) \cup \delta_2(q, s)$. FSA are $\Cup$-closed for finite sets, and the following result shows that $\Cup$ approximates any upper bound with respect to the ordering $\sqsubseteq_{\mathfrak{F}}$.

Lemma 1. *Given two FSA M_1 and M_2 we have: $\mathscr{L}(M_1) \cup \mathscr{L}(M_2) \subseteq \mathscr{L}(M_1 \Cup M_2)$.*

We can now define $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!] : \mathfrak{F} \rightarrow \mathfrak{F}$ as follows: $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!](M) = \mathring{\alpha}(P_0) \Cup M \Cup (\Cup\{M' \mid M' \in \mathcal{T}^\sharp(M)\})$. Observe that the set of possible successors of a given automaton M, i.e., $\mathcal{T}^\sharp(M)$, is finite since we have a (finite family of) successor for every transition edge of M and M has a finite set of edges. Since FSA are $\Cup$-closed for finite sets, then $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ is well defined. Let $\wp_F(\mathfrak{F}^*)$ denote the domain of finite sets of strings of FSA and let us define function $\alpha_S : \wp_F(\mathfrak{F}^*) \rightarrow \mathfrak{F}$ as $\alpha_S(M_0 \ldots M_k) = \Cup\{M_i \mid 0 \leq i \leq k\}$ and $\alpha_S(K) = \Cup\{\alpha_S(M_0 \ldots M_k) \mid M_0 \ldots M_k \in K\}$, with $K \in \wp_F(\mathfrak{F}^*)$. Function α_S is additive and it defines a GC $(\wp_F(\mathfrak{F}^*), \alpha_S, \gamma_S, \mathfrak{F})$. The following result shows that, when considering finite sets of sequences of FSA, $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ correctly approximates $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$ on $\mathfrak{F}$.

Theorem 7. *For any $K \in \wp_F(\mathfrak{F}^*)$ we have $\alpha_S(\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!](K)) \sqsubseteq_{\mathfrak{F}} \mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!](\alpha_S(K))$.*

The domain $\langle \mathfrak{F}, \sqsubseteq_{\mathfrak{F}} \rangle$ has infinite ascending chains, which means that, in general, the fixpoint computation of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ on $\mathfrak{F}$ may not converge. A typical solution for this situation is the use of a widening operator which forces convergence towards an upper approximation of all intermediate computations along the fixpoint iteration, i.e., an element in $\mathfrak{F}$ which upper approximates the iterates of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$. We refer to the widening operation over FSA described by D'Silva [14]. Here the widening operator between two FSA $M_1 = (Q_1, E_1, S_1)$ and $M_2 = (Q_2, E_2, S_2)$ over a finite alphabet A is formalized in terms of an equivalence relation $R \subseteq Q_1 \times Q_2$ between states. R, also called *widening seed*, is used to define another equivalence relation $\equiv_R \subseteq Q_2 \times Q_2$ over the states of M_2, such that $\equiv_R = R \circ R^{-1}$. The widening between M_1 and M_2 is then given by the quotient of M_2 with respect to the partition induced by $\equiv_R$: $M_1 \triangledown M_2 = M_2/\equiv_R$. By changing the widening seed we obtain different widening operators. It has been proved that convergence is guaranteed when the widening seed is the relation $R_n \subseteq Q_1 \times Q_2$ such that $(q_1, q_2) \in R_n$ if q_1 and q_2 recognize the same language of strings of length at most n [14]. When considering the widening seed R_n we have that two states q and q' of M_2 are $\equiv_{R_n}$-equivalent if they recognize the same language of length at most n that is recognized by a state r of M_1, i.e., if $\exists r \in Q_1 : (r, q) \in R_n$ and $(r, q') \in R_n$. $\triangledown_n$ denotes the widening operator that uses R_n as widening seed. $\triangledown_n$ is well defined if $\mathring{\mathbb{I}}$ is finite. This can be achieved by considering expressions as terms and by applying some of the standard methods for approximating them. The most straightforward one is the depth-k string abstraction [24], while more refined expression abstractions can be designed by considering graph-based or grammar-based term abstractions [3, 15]. For

simplicity we consider here the depth-k term abstraction where expressions are represented as trees with leafs that are natural numbers denoting either a memory location or a constant, and internal nodes are the operators constructing expressions, namely the unary operator MEM or the binary operators op. We annotate each node with its depth, namely with the length of the path from the root to the node. The depth-k abstraction, given a tree representation of an expression, considers only the nodes with depth less or equal to k and "cuts" the remaining nodes by approximating them with $\top$. For example, the depth-3 abstraction of expression $\texttt{MEM}[(\texttt{MEM}[a] \ \texttt{op} \ \texttt{MEM}[b \ \texttt{op} \ \texttt{MEM}[c]]) \ \texttt{op} \ d]$ is $\texttt{MEM}[(\texttt{MEM}[\top] \ \texttt{op} \ \texttt{MEM}[\top]) \ \texttt{op} \ d]$. Given $k \in \mathbb{N}$, let $\iota_k : \mathring{\mathbb{I}} \rightarrow \mathring{\mathbb{I}}_k$ be the instruction abstraction that applies the depth-k abstraction to the expressions occurring in an abstract instruction, and let $\alpha_k : \mathfrak{F} \rightarrow \mathfrak{F}_k$ be the function that abstracts the edge labels of a FSA in $\mathfrak{F}$ according to ι_k. It is possible to show that $(\mathfrak{F}, \alpha_k, \gamma_k, \mathfrak{F}_k)$ is a GC, where $\gamma_k(M^k) = \Cup\{M' \mid \alpha_k(M') \sqsubseteq_{\mathfrak{F}} M^k\}$. This allows us to approximate the least fixpoint of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ on $\langle \mathfrak{F}_k, \sqsubseteq_{\mathfrak{F}} \rangle$ with the limit $\mathbf{W}[\![P_0]\!]$ of the following widening sequence: $W_0 = \alpha_k(\mathring{\alpha}(P_0))$ and $W_{i+1} = W_i \ \triangledown_n \ \alpha_k(\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!](\gamma_k(W_i)))$. Let us refer to $\mathbf{W}[\![P_0]\!]$ as the *widened fixpoint* of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$ and to $W_0 W_1, \ldots$ as the *widening sequence* of $\mathcal{F}^{\Cup}_{\mathcal{T}^\sharp}[\![P_0]\!]$. From the correctness of $\triangledown_n$ and by Theorem 7, it follows that the widening sequence $W_0 W_1 \ldots$ converges to an upper-approximation of the least fixpoint of $\mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!]$, namely any automata modelling a possible static variant of P_0 is approximated by $\mathbf{W}[\![P_0]\!]$ i.e., $\ldots M_i \ldots \in \mathit{lfp}^{\subseteq} \mathcal{F}_{\mathcal{T}^\sharp}[\![P_0]\!] \ \Rightarrow \ M_i \sqsubseteq_{\mathfrak{F}} \mathbf{W}[\![P_0]\!]$. Therefore $\mathscr{L}(\mathbf{W}[\![P_0]\!])$ contains all the possible sequences of abstract instructions that can be executed by a metamorphic variant of P_0. As a consequence, a program Q is a regular (abstract) metamorphic variant of P_0 if $\mathbf{W}[\![P_0]\!]$ recognizes all the sequences of abstract instructions that correspond to the runs of Q up to address renaming: $P_0 \leadsto_{\mathfrak{F}_k} Q$ iff there exists an address renaming ϑ such that $\vartheta(\mathscr{L}(\alpha_k(\mathring{\alpha}(Q)))) \subseteq \mathscr{L}(\mathbf{W}[\![P_0]\!])$. The language $\mathscr{L}(\mathbf{W}[\![P_0]\!])$ represents the regular metamorphic signature of P_0 and the automaton $\mathbf{W}[\![P_0]\!]$ represents the mechanism of generation of the metamorphic variants and therefore it provides a model of the metamorphic engine of P_0. Fig. 6 (a) shows the widened fixpoint $\mathbf{W}[\![P_0]\!]$ of program P_0 in Fig. 2, where the widening seed is R_2 and $k \geq 3$, This automaton recognizes any possible program that can be obtained during the execution of P_0. Note that, we may have false positives, as for example the sequences of instructions along the bold path $\texttt{MEM}[f] := 100; \texttt{input} \Rightarrow \texttt{MEM}[a]; \texttt{MEM}[a] \bmod 2 = 0; \texttt{MEM}[b] := \texttt{MEM}[a]; \texttt{goto}; \texttt{MEM}[b] := \texttt{MEM}[a]; \texttt{goto}; \ldots$ which is not a run of any of the variants of P_0. Regular metamorphism can easily cope with metamorphic transformations commonly used by malware (e.g., Win95/Regswap, Win32/Ghost, Win95/Zperm, Win95/Zmorph, Win32/Evol) such as: *register swap* that changes the registers used by the program; *code permutation* that changes the order in which instructions appear in memory while preserving their execution order through the insertion of direct jumps; *junk/nop insertion* that inserts junk instructions and semantic-nops, namely instructions that are not executed or that do not alter program functionality. Observe that all these transformations can be seen as special cases of *code substitution*. Let P_0 be a metamorphic malware: whenever a sequence s_1 of instructions is substituted with an equivalent one s_2, we have that during the widened fixpoint computation a new path containing sequence s_2 is added to the widened fixpoint $\mathbf{W}[\![P_0]\!]$. Therefore, by correctness, $\mathbf{W}[\![P_0]\!]$ recognizes all the possible metamorphic variants of P_0 obtained

through code substitution. Of course it is possible to further abstract $\mathbf{W}[\![P_0]\!]$ in order to address semantic-nop/junk insertion, permutation and register swap in a more efficient way, namely in such a way that the resulting widened fixpoint is an automaton of a reduced size. In semantic-nop insertion, the more precise is the static analysis used for identifying (sequences of) instructions that are equivalent to `nop`, the smaller is the widened fixpoint $\mathbf{W}[\![P_0]\!]$ that we obtain. In code permutation, a smaller FSA can be obtained by performing `goto`-reduction, i.e., by folding nodes reachable by `goto`-instructions. In register swapping it is sufficient to replace registers names (i.e., memory locations) with uninterpreted symbols and then use unification to bind these uninterpreted symbols to the actual register names (i.e., memory locations) as done in [5]. Let us consider program P_0^+ obtained by enriching the metamorphic engine of program P_0 of Fig. 2 with a code permutation and a transformation that substitutes instruction $\texttt{MEM}[e_1] := e_2$ with the equivalent sequence $\texttt{push}\ e_1, \texttt{pop}\ e_2$. A possible evolution is shown below, where ME denotes the metamorphic engine.

```
P0+ :
  1 : goto 8
  2 : if (MEM[a] mod 2) goto 11
  3 : nop
  4 : goto 100
  5 : push MEM[a]/2
  6 : pop a
  7 : goto 12
  8 : MEM[f] := 100
  9 : input => MEM[a]
 10 : goto 2
 11 : MEM[a] := (MEM[a] + 1)/2
 12 : ME
 13 : goto 9
100 : push MEM[a]
101 : pop b
102 : goto 5
```

Fig. 6 (b) shows the FSA that represents an approximation of all the possible evolutions of program P_0^+ when $k \geq 3$. This FSA is obtained through widening with widening seed R_2 and by applying the `goto`-reduction to handle permutation. We can observe that every time that in the automaton in Fig. 6 (b) we have an edge labeled with $\texttt{MEM}[e_1] := e_2$ between two states q and p, then we also have a path labeled with $\texttt{push}\ e_2, \texttt{pop}\ e_1$ that connects q and p, and this precisely captures the fact that the metamorphic engine implements this substitution. The `goto`-reduction allows here to have a reduced FSA, and the self-loop labeled with `nop` makes clear that the metamorphism could insert an unbounded number of `nop` instructions.

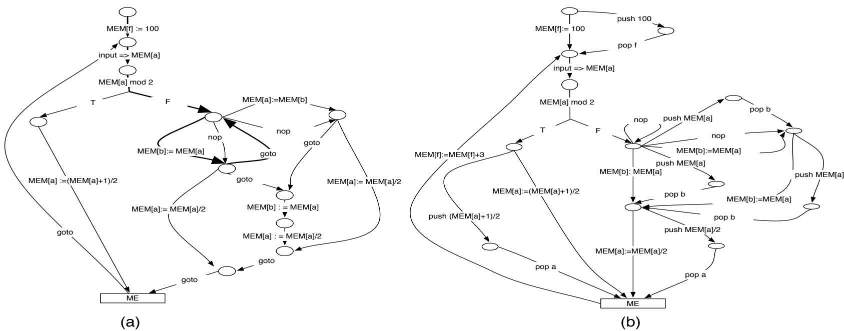


Fig 6. Widened phase semantics

6 Related Works and Discussion

In [13] the authors use trace semantics to characterize the behaviours of both the malware and the potentially infected program, and use abstract interpretation to “hide” their

irrelevant behaviours. A program is infected by a malware if their behaviours are indistinguishable up to a certain abstraction, which corresponds to some obfuscations. A significant limitation of this work is that the knowledge of the obfuscation is essential in order to derive abstractions. In [19] the authors model the malware M as a formula in the new logic CTPL, which is an extension of CTL able to handle register renaming. A program P is infected by M, if P satisfies the CTPL formula that models M. By knowing the obfuscations used by malware M it is possible to design CTPL specifications that recognise several metamorphic variants of M. In [7] the idea is to model the malware as a template that expresses the malicious intent. Also in this case the definition of the template is driven by the knowledge of the obfuscations commonly used by malware. Some researchers have tried to detect metamorphic malware by modelling the metamorphic engine as formal grammars and automata [25, 20, 16]. These works are promising, but the design of the grammar and automata is based on the knowledge of the metamorphic transformations used, and none of them provides a methodology for extracting a grammar or an automata from a given metamorphic malware. To the best of our knowledge, we are not aware of any work modelling metamorphism without any a priori knowledge of the transformations used by the metamorphic engine. The only other work we are aware of that formally addresses the analysis of self-modifying code is the one of Cai et al. [4]. However, their goals and results are very different from ours: Cai et al. propose a general framework based on Hoare logic to verify self-modifying code, while we use program semantics and abstract interpretation to extract metamorphic signature from malicious self-modifying code. In this sense, our key contribution relies upon the idea that abstract interpretation of phase semantics may provide useful information about the way code changes, i.e., about the metamorphic engine itself. Interestingly, the language recognized by $\mathbf{W}[\![P]\!]$ provides an upper-approximation of the possible metamorphic variants of the original malware, while the automaton itself models the mechanism of generation of such variants, i.e., the metamorphic engine. With our approach it is therefore possible to extract properties of the implementation of the metamorphic engine by abstract interpretation of the phase semantics. It is clear that the depth-k abstraction considered here for approximating the language of instructions towards a finite alphabet for widening traces of FSA is for sake of simplicity. In general, widening phases for taming the sequence of modified programs (FSA) generated by metamorphism into a single FSA modeling regular metamorphism may require a notion of *higher-order widening* on FSA, acting both at the level of the graph-structure of the FSA, for approximating the language of instructions, and at the level of the instruction set, for approximating the way a single instruction may be composed. The abstraction of code layout may induce the abstraction of instructions, which itself can be solved by means of FSA. This opens an interesting new field that may represent a future challenge for abstract interpretation: *the abstraction of code layout*, where the code is the object of abstraction and the way it is generated is the object of abstract interpretation. Of course FSA provide just regular language-based abstractions of the metamorphic engine. More sophisticated approximations, using for instance *a la Cousot*'s context free grammars and set-constraint-based abstractions of sequences of binary instructions [10], may provide alternative and effective solutions for non-regular metamorphism.

References

1. Balakrishnan, G., Gruian, R., Reps, T.W., Teitelbaum, T.: Codesurfer/x86-a platform for analyzing x86 executables. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 250–254. Springer, Heidelberg (2005)
2. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
3. Bruynooghe, M., Janssens, G., Callebaut, A., Demoen, B.: Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In: Proc. Symposium on Logic Programming, pp. 192–204 (1987)
4. Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: Proc. ACM Conf. on Programming Language Design and Implementation (PLDI 2007), pp. 66–77 (2007)
5. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proc. USENIX Security Symp., pp. 169–186 (2003)
6. Christodorescu, M., Jha, S.: Testing malware detectors. In: Proc. ACM SIGSOFT Internat. Symp. on Software Testing and Analysis (ISSTA 2004), pp. 34–44 (2004)
7. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proc. IEEE Security and Privacy 32–46 (2005)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. ACM Symp. on Principles of Programming Languages (POPL 1977), pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. ACM Symp. on Principles of Programming Languages (POPL 1979), pp. 269–282 (1979)
10. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, pp. 170–181 (1995)
11. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. 277(1-2), 47–103 (2002)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. ACM Symp. on Principles of Programming Languages, POPL 1978 (1978)
13. Dalla Preda, M., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. ACM Trans. Program. Lang. Syst. 30(5), 1–54 (2008)
14. D'Silva, V.: Widening for automata. Diploma Thesis, Institut Fur Informatick, Universitat Zurich (2006)
15. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proc. ACM Conf. Programming Language Design and Implementation, pp. 242–256 (1994)
16. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. In: Proc. World Academy of Science, Engineering and Technology (PWASET), vol. 20 (2007)
17. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. of the ACM. 47(2), 361–416 (2000)
18. Holzer, A., Kinder, J., Veith, H.: Using verification technology to specify and detect malware. In: Moreno Díaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST 2007. LNCS, vol. 4739, pp. 497–504. Springer, Heidelberg (2007)
19. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
20. Qozah. Polymorphism and grammars. 29A E-zine (2009)
21. Singh, P., Lakhotia, A.: Static verification of worm and virus behaviour in binary executables using model checking. In: Proc. IEEE Information Assurance Workshop (2003)

22. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional, Reading (2005)
23. Ször, P., Ferrie, P.: Hunting for metamorphic. In: Proc. Virus Bulleting Conference, pp. 123–144. Virus Bulletin Ltd. (2001)
24. Tamaki, H., Sato, T.: Program Transformation Through Meta-shifting. New Generation Computing 1(1), 93–98 (1983)
25. Zbitskiy, P.: Code mutation techniques by means of formal grammars and automatons. Journal in Computer Virology (2009)

Small Formulas for Large Programs: On-Line Constraint Simplification in Scalable Static Analysis*

Isil Dillig**, Thomas Dillig, and Alex Aiken

Department of Computer Science,
Stanford University
{isil,tdillig,aiken}@cs.stanford.edu

Abstract. Static analysis techniques that represent program states as formulas typically generate a large number of redundant formulas that are incrementally constructed from previous formulas. In addition to querying satisfiability and validity, analyses perform other operations on formulas, such as quantifier elimination, substitution, and instantiation, most of which are highly sensitive to formula size. Thus, the scalability of many static analysis techniques requires controlling the size of the generated formulas throughout the analysis. In this paper, we present a practical algorithm for reducing SMT formulas to a *simplified form* containing no redundant subparts. We present experimental evidence that on-line simplification of formulas dramatically improves scalability.

1 Introduction

Software verification techniques have benefited greatly from recent advances in SAT and SMT solving by encoding program states as formulas and determining the feasibility of these states by querying satisfiability. Despite tremendous progress in solving SAT and SMT formulas over the last decade [1–8], the scalability of many software verification techniques relies crucially on controlling the size of the formulas generated by the analysis, because many of the operations performed on these formulas are highly sensitive to formula size. For this reason, much research effort has focused on identifying only those states and predicates relevant to some property of interest. For example, *predicate abstraction*-based approaches using *counter-example guided abstraction refinement* [9–11] attempt to discover a small set of predicates relevant to verifying a property and only include this small set of predicates in their formulas. Similarly, many path-sensitive static analysis techniques have successfully employed various heuristics to identify which path conditions are likely to be relevant for some property of interest.

* This work was supported by grants from NSF (CNS-050955, CCF-0430378) with additional support from DARPA.

** Supported by the Stanford Graduate Fellowship.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 236–252, 2010.
© Springer-Verlag Berlin Heidelberg 2010

For example, *property simulation* only tracks those branch conditions for which the property-related behavior differs along the arms of the branch [12]. Other path-sensitive analysis techniques attempt to improve their scalability by either only tracking path conditions intraprocedurally or by heuristically selecting a small set of predicates to track across function boundaries [13, 14].

All of these different techniques share one important underlying assumption that has been validated by a large body of empirical evidence: Many program conditions do not matter for verifying most properties of interest, making it possible to construct much smaller formulas sufficient to prove the property. If this is indeed the case, then one might suspect that even if we construct a formula ϕ characterizing some program property P without being particularly careful about what conditions to track, it should be possible to use ϕ to construct a much smaller, equivalent formula ϕ' for P since many predicates used in ϕ do not affect P's truth value.

In this paper, we present a systematic and practical approach for simplifying formulas that identifies and removes irrelevant predicates and redundant subexpressions as they are generated by the analysis. In particular, given an input formula ϕ, our technique produces an equivalent formula ϕ' such that no simpler equivalent formula can be obtained by replacing any subset of the *leaves* (i.e., syntactic occurrences of atomic formulas) used in ϕ' by *true* or *false*. We call such a formula ϕ' *simplified*.

Like all the afore-mentioned approaches to program verification, our interest in simplification is motivated by the goal of generating formulas small enough to make software verification scalable. However, we attack the problem from a different angle: Instead of restricting the set of predicates that are allowed to appear in formulas, we continuously simplify the constraints generated by the analysis. This approach has two advantages: First, it does not require heuristics to decide which predicates are relevant, and second, this approach removes all redundant subparts of a formula in addition to filtering out irrelevant predicates.

To be concrete, consider the following code snippet:

```
enum op_type {ADD=0, SUBTRACT=1, MULTIPLY=2, DIV=3};
int perform_op(op_type op, int x, int y) {
   int res;
   if(op == ADD) res = x+y;
   else if(op == SUBTRACT) res = x-y;
   else if(op == MULTIPLY) res = x*y;
   else if(op == DIV) { assert(y!=0); res = x/y; }
   else res = UNDEFINED;
   return res; }
```

The `perform_op` function is a simple evaluation procedure inside a calculator program that performs a specified operation on `x` and `y`. This function aborts if the specified operation is division and the divisor is 0. Assume we want to know the constraint under which the function returns, i.e., does not abort. This constraint is given by the disjunction of the constraints under which each branch

of the `if` statement does not abort. The following formula, constructed in a straightforward way from the program, describes this condition:

$$op = 0 \vee (op \neq 0 \wedge op = 1) \vee (op \neq 0 \wedge op \neq 1 \wedge op = 2) \vee$$
$$(op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op = 3 \wedge y \neq 0) \vee$$
$$(op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op \neq 3)$$

Here, each disjunct is associated with one branch of the `if` statement. In each disjunct, a disequality constraint of the form $op \neq 0, op \neq 1, \ldots$ states that the previous branches were not taken, encoding the semantics of an `else` statement. In the fourth disjunct, the additional constraint $y \neq 0$ encodes that if this branch is taken, `y` cannot be 0 for the function to return.

While this automatically generated constraint faithfully encodes the condition under which the function returns, it is far from concise. In fact, the above constraint is equivalent to the much simpler formula:

$$op \neq 3 \vee y \neq 0$$

This formula is in *simplified form* because it is equivalent to the original formula and replacing any of the remaining leaves by *true* or *false* would not result in an equivalent formula. This simpler constraint expresses exactly what is relevant to the function's return condition and makes no reference to irrelevant predicates, such as $op = 0, op = 1$, and $op = 2$. Although the original formula corresponds to a brute-force enumeration of all paths in this function, its simplified form yields the most concise representation of the function's return condition without requiring specialized techniques for identifying relevant predicates.

The rest of the paper is organized as follows: Section 2 introduces preliminary definitions. Section 3 defines simplified form and highlights some of its properties. Section 4 presents a practical simplification algorithm, and Section 5 describes simplification in the context of program analysis. Section 6 reports experimental results, and Section 7 surveys related work. To summarize, this paper makes the following key contributions:

- We present an *on-line constraint simplification* algorithm for improving SMT-based static analysis techniques.
- We define what it means for a formula to be in *simplified form* and detail some important properties of this form.
- We give a practical algorithm for reducing formulas to their simplified form and show how this algorithm naturally integrates into the DPLL($\mathcal{T}$) framework for solving SMT formulas.
- We demonstrate the effectiveness of our on-line simplification algorithm in the context of a program verification framework and show that simplification improves overall performance by orders of magnitude, often allowing analysis runs that did not terminate within the allowed resource limits to complete in just a few seconds.

2 Preliminaries

Any quantifier-free formula $\phi_{\mathcal{T}}$ in theory $\mathcal{T}$ is defined by the following grammar:

$$\phi_{\mathcal{T}} := \mathit{true} \mid \mathit{false} \mid A_{\mathcal{T}} \mid \neg A_{\mathcal{T}} \mid \phi'_{\mathcal{T}} \wedge \phi''_{\mathcal{T}} \mid \phi'_{\mathcal{T}} \vee \phi''_{\mathcal{T}}$$

In the above grammar, $A_{\mathcal{T}}$ represents an *atomic formula* in theory $\mathcal{T}$, such as the boolean variable x in propositional logic or the inequality $a + 2b \leq 3$ in linear arithmetic. Observe that the above grammar requires formulas to be in *negation normal form (NNF)* because only atomic formulas may be negated. While the rest of this paper relies on formulas being in NNF, this restriction is not important since any formula may be converted to NNF using De Morgan's laws in linear time without increasing the size of the formula (see Definition 2).

Definition 1. (Leaf) *We refer to each occurrence of an atomic formula $A_{\mathcal{T}}$ or its negation $\neg A_{\mathcal{T}}$ as a* leaf *of the formula in which it appears.*

It is important to note that different occurrences of the same (potentially negated) atomic formula in $\phi_{\mathcal{T}}$ form distinct leaves. For example, the two occurrences of $f(x) = 1$ in $f(x) = 1 \vee (f(x) = 1 \wedge x + y \leq 1)$ correspond to two distinct leaves. Also, observe that leaves are allowed to be negations. For instance, in the formula $\neg(x = y)$, $(x = y)$ is not a leaf; the only leaf of the formula is $\neg(x = y)$.

In the rest of this paper, we restrict our focus to quantifier-free formulas in theory $\mathcal{T}$, and we assume there is a decision procedure $D_{\mathcal{T}}$ that can be used to decide the satisfiability of a quantifier-free formula $\phi_{\mathcal{T}}$ in theory $\mathcal{T}$. Where irrelevant, we omit the subscript $\mathcal{T}$ and denote formulas by ϕ.

Definition 2. (Size) *The* size *of a formula ϕ is the number of leaves ϕ contains.*

Definition 3. (Fold) *The* fold *operation removes constant leaves (i.e.,* true*,* false*) from the formula. In particular,* Fold(ϕ) *is a formula ϕ' such that (i) $\phi \Leftrightarrow \phi'$, (ii) ϕ' is just* true *or* false *or ϕ' mentions neither* true *nor* false*.*

It is easy to see that it is possible to construct this fold operation such that it reduces the size of the formula ϕ at least by one if ϕ contains *true* or *false* but ϕ is not initially *true* or *false*.

3 Simplified Form

In this section, we first define *redundancy* and describe what it means for a formula to be in *simplified form*. We then highlight some important properties of simplified forms. Notions of redundancy similar to ours have been studied in other contexts, such as in *automatic test pattern generation* and *vacuity detection*; see Section 7 for a discussion.

Definition 4. ($\boldsymbol{\phi^+}$(L), $\boldsymbol{\phi^-}$(L)) *Let ϕ be a formula and let L be a leaf of ϕ. $\phi^+(L)$ is obtained by replacing L by* true *and applying the fold operation. Similarly, $\phi^-(L)$ is obtained by replacing L by* false *and folding the resulting formula.*

Example 1. Consider the formula:

$$\underbrace{x = y}_{L_0} \wedge (\underbrace{f(x) = 1}_{L_1} \vee (\underbrace{f(y) = 1}_{L_2} \wedge \underbrace{x + y \leq 1}_{L_3}))$$

Here, $\phi^+(L_1)$ is $(x = y)$, and $\phi^-(L_2)$ is $(x = y \wedge f(x) = 1)$.

Observe that for any formula ϕ, $\phi^+(L)$ is an overapproximation of ϕ, i.e., $\phi \Rightarrow \phi^+(L)$, and $\phi^-(L)$ is an underapproximation, i.e., $\phi^-(L) \Rightarrow \phi$. This follows immediately from Definition 4 and the monotonicity of NNF. Also, by construction, the sizes of $\phi^+(L)$ and $\phi^-(L)$ are at least one smaller than the size of ϕ.

Definition 5. (Redundancy) *We say a leaf L is* non-constraining *in formula ϕ if $\phi^+(L) \Rightarrow \phi$ and* non-relaxing *if $\phi \Rightarrow \phi^-(L)$. Leaf L is* redundant *if L is either non-constraining or non-relaxing.*

The following corollary follows immediately from definition:

Corollary 1. *If a leaf L is non-constraining, then $\phi \Leftrightarrow \phi^+(L)$, and if L is non-relaxing, then $\phi \Leftrightarrow \phi^-(L)$.*

Intuitively, if replacing a leaf L by *true* in formula ϕ results in an equivalent formula, then L does not constrain ϕ; hence, we call such a leaf non-constraining. A similar intuition applies for non-relaxing leaves.

Example 2. Consider the formula from Example 1. In this formula, leaves L_0 and L_1 are not redundant, but L_2 is redundant because it is non-relaxing. Leaf L_3 is both non-constraining and non-relaxing, and thus also redundant.

Note that if two leaves L_1 and L_2 are redundant in formula ϕ, this does not necessarily mean we can obtain an equivalent formula by replacing both L_1 and L_2 with *true* (if non-constraining) or *false* (if non-relaxing). This is the case because eliminating L_1 may render L_2 non-redundant and vice versa.

Definition 6. (Simplified Form) *We say a formula ϕ is in* simplified form *if no leaf mentioned in ϕ is redundant.*

Lemma 1. *If a formula ϕ is in simplified form, replacing any subset of the leaves used in ϕ by* true *or* false *does not result in an equivalent formula.*

Proof. The proof is by induction. If ϕ contains a single leaf, the property trivially holds. Suppose ϕ is of the form $\phi_1 \vee \phi_2$. Then, if ϕ has a simplification $\phi_1' \vee \phi_2'$ where both ϕ_1' and ϕ_2' are simplified, then either $\phi_1' \vee \phi_2$ or $\phi_1 \vee \phi_2'$ is also equivalent to ϕ. This is the case because $(\phi \Leftrightarrow \phi_1' \vee \phi_2') \wedge (\phi \not\Leftrightarrow \phi_1' \vee \phi_2) \wedge (\phi \not\Leftrightarrow \phi_1 \vee \phi_2')$ is unsatisfiable. A similar argument applies if the connective is $\wedge$. $\square$

The following corollary follows directly from Lemma 1:

Corollary 2. *A formula ϕ in simplified form is satisfiable if and only if it is not syntactically* false *and valid if and only if it is syntactically* true.

This corollary is important in the context of on-line simplification in program analysis because, if formulas are kept in simplified form, then determining satisfiability and validity becomes just a syntactic check.

Observe that while a formula ϕ in simplified form is guaranteed not to contain redundancies, there may still exist a smaller formula ϕ' equivalent to ϕ. In particular, a non-redundant formula may be made smaller, for example, by factoring common subexpressions. We do not address this orthogonal problem in this paper, and the algorithm given in Section 4 does not change the structure of the formula.

Example 3. Consider the propositional formula $(a \wedge b) \vee (a \wedge c)$. This formula is in simplified form, but the equivalent formula $a \wedge (b \vee c)$ contains fewer leaves.

As this example illustrates, it is not possible to determine the equivalence of two formulas by checking whether their simplified forms are syntactically identical. Furthermore, as illustrated by the following example, the simplified form of a formula ϕ is not always guaranteed to be unique.

Example 4. Consider the formula $x = 1 \vee x = 2 \vee (1 \leq x \wedge x \leq 2)$ in the theory of linear integer arithmetic. The two formulas $x = 1 \vee x = 2$ and $1 \leq x \wedge x \leq 2$ are both simplified forms that can be obtained from the original formula.

Lemma 2. *If ϕ is a formula in simplified form, then* $\mathrm{NNF}(\neg\phi)$ *is also in simplified form, where* NNF *converts the formula to negation normal form.*

Proof. Suppose $\mathit{NNF}(\neg\phi)$ was not in simplified form. Then, it would be possible to replace one leaf, say L, by *true* or *false* to obtain a strictly smaller, but equivalent formula. Now consider negating the simplified form of $\mathit{NNF}(\neg\phi)$ to obtain ϕ' which is equivalent to ϕ. Note that the $\neg L$ is a leaf in ϕ, but not in ϕ'. Thus, ϕ could not have been in simplified form. □

Hence, if a formula is in simplified form, then its negation does not need to be resimplified, an important property for on-line simplification in program analysis. However, simplified forms are not preserved under conjunction or disjunction.

Lemma 3. *For every formula ϕ, there exists a formula ϕ' in simplified form such that (i) $\phi \Leftrightarrow \phi'$, and (ii)* $\mathrm{size}(\phi') \leq \mathrm{size}(\phi)$.

Proof. Consider computing ϕ' by checking every leaf L of ϕ for redundancy and replacing L by *true* if it is non-constraining and by *false* if it is non-relaxing. If this process is repeated until there are no redundant leaves, the resulting formula is in simplified form and contains at most as many leaves as ϕ. □

The above lemma states that converting a formula to its simplified form never increases the size of the formula. This property is desirable because, unlike other representations like BDDs that attempt to describe the formula compactly, computing a simplified form is guaranteed not to cause a worst-case blow-up. In the experience of the authors, this property is crucial in program verification.

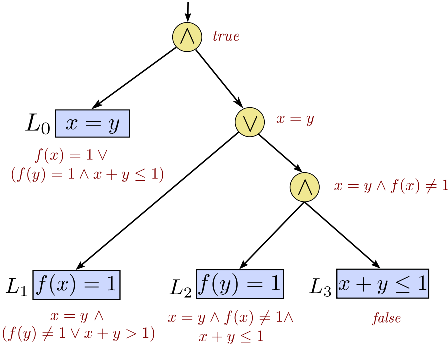


Fig. 1. The representation of the formula from Example 1. The critical constraint at each node is shown in red. Observe that the critical constraint for L_3 is *false*, making L_3 both non-constraining and non-relaxing. The critical constraint of L_2 implies its negation; hence, L_2 is non-relaxing.

4 Algorithm to Compute Simplified Forms

While the proof of Lemma 3 sketches a naive way of computing the simplified form of a formula ϕ, this approach is suboptimal because it requires repeatedly checking the satisfiability of a formula twice as large as ϕ until no more redundant leaves can be identified. In this section, we present a practical algorithm to compute simplified forms. For convenience, we assume formulas are represented as trees; however, the algorithm is easily modified to work on directed acyclic graphs, and in fact, our implementation uses DAGs to represent formulas. A node in the tree represents either an $\wedge$ or $\vee$ connective or a leaf. We assume connectives have at least two children but may have more than two.

4.1 Basic Algorithm

Recall that a leaf L is non-constraining if and only if $\phi^+(L) \Rightarrow \phi$ and non-relaxing if and only if $\phi \Rightarrow \phi^-(L)$. Since the size of $\phi^+(L)$ and $\phi^-(L)$ may be only one less than ϕ, checking whether L is non-constraining or non-relaxing using Definition 5 requires checking the validity of formulas twice as large as ϕ.

A key idea underlying our algorithm is that it is possible to check for redundancy of a leaf L by checking the validity of formulas no larger than ϕ. In particular, for each leaf L, our algorithm computes a formula $\alpha(L)$, called the *critical constraint* of L, such that (i) $\alpha(L)$ is no larger than ϕ, (ii) L is non-constraining if and only if $\alpha(L) \Rightarrow L$, and (iii) L is non-relaxing if and only if $\alpha(L) \Rightarrow \neg L$. This allows us to determine whether each leaf is redundant by determining the satisfiability of formulas no larger than the original formula ϕ.

Definition 7. (Critical constraint)

- *Let R be the root node of the tree. Then,* $\alpha(R) = \text{true}$.
- *Let N be any node other than the root node. Let P denote the parent of N in the tree, and let $S(N)$ denote the set of siblings of N. Let $\star$ denote $\neg$ if P is an $\vee$ connective, and nothing if P is an $\wedge$ connective. Then,*

$$\alpha(N) = \alpha(P) \wedge \bigwedge_{S_i \in S(N)} \star S_i$$

Intuitively, the critical constraint of a leaf L describes the condition under which L will be relevant for either permitting or disallowing a particular model of ϕ. Clearly, if the assignment to L is to determine whether ϕ is *true* or *false* for a given interpretation, then all the children of an $\wedge$ connective must be true if this $\wedge$ node is an ancestor of L; otherwise ϕ is already false regardless of the assignment to L. Also, observe that L is not relevant in permitting or disallowing a model of ϕ if some other path not involving L is satisfied because ϕ will already be true regardless of the truth value of L. Hence, the critical constraint includes the negation of the siblings at an $\vee$ connective while it includes the siblings themselves at an $\wedge$ node. The critical constraint can be viewed as a *context* in the general framework of *contextual rewriting* [15, 16]; see Section 7 for a discussion.

Example 5. Figure 1 shows the representation of the formula from Example 1 along with the critical constraints of each node.

Lemma 4. *A leaf L is non-constraining if and only if $\alpha(L) \Rightarrow L$.*

Proof. (Sketch) Suppose $\alpha(L) \Rightarrow L$, but L is constraining, i.e., the formula $\gamma = (\phi^+(L) \wedge \neg\phi)$ is satisfiable. Then, there must exist some model M of γ that satisfies $\phi^+(L)$ but not ϕ. For M to be a model of $\phi^+(L)$ but not ϕ, it must (i) assign all the children of any $\wedge$ node that is an ancestor of L to *true*, (ii) it must assign L to *false*, and (iii) it must assign any other children of an $\vee$ node that is an ancestor of L to *false*. By (i) and (iii), such a model must also satisfy $\alpha(L)$. Since $\alpha(L) \Rightarrow L$, M must also satisfy L, contradicting (ii). The other direction is analogous. □

Lemma 5. *A leaf L is non-relaxing if and only if $\alpha(L) \Rightarrow \neg L$.*

Proof. Similar to the proof of Lemma 4.

We now formulate a simple recursive algorithm, presented in Figure 2, to reduce a formula ϕ to its simplified form. In this algorithm, N is a node representing the current subpart of the formula, and α denotes the critical constraint associated with N. If C is some ordered set, we use the notation $C_{<i}$ and $C_{>i}$ to denote the set of elements before and after index i in C respectively. Finally, we use the notation $\star$ as in Definition 7 to denote $\neg$ if the current node is an $\vee$ connective and nothing otherwise.

Observe that, in the algorithm of Figure 2, the critical constraint of each child c_i of a connective node is computed by using the new siblings c'_k that have been simplified. This is crucial for the correctness of the algorithm because, as pointed out in Section 3, if two leaves L_1 and L_2 are both initially redundant, it does not mean L_2 stays redundant after eliminating L_1 and vice versa. Using the simplified siblings in computing the critical constraint of c_i has the same effect as rechecking whether c_i remains redundant after simplifying sibling c_k.

Another important feature of the algorithm is that, at connective nodes, each child is simplified as long as any of their siblings change, i.e., the recursive invocation returns a new sibling not identical to the old one. The following example illustrates why this is necessary.

simplify(N, α)

- If N is a leaf:
  - If $\alpha \Rightarrow N$ return *true*
  - If $\alpha \Rightarrow \neg N$ return *false*
  - Otherwise return N
- If N is a connective, let C denote the ordered set of children of N, and let C' denote the new set of children of N .
  - For each $c_i \in C$:

$$\begin{aligned} \alpha_i &= \alpha \ \wedge \ (\textstyle\bigwedge_{c_j \in C_{>i}} \star c_j) \ \wedge (\textstyle\bigwedge_{c'_k \in C'_{<i}} \star c'_k) \\ c'_i &= \text{simplify}(c_i, \alpha_i) \\ C' &= C' \cup c'_i \end{aligned}$$

  - Repeat the previous step until $\forall i . c_i = c'_i$
  - If N is an $\wedge$ connective, return $\bigwedge_{c'_i \in C'} c'_i$
  - If N is an $\vee$ connective, return $\bigvee_{c'_i \in C'} c'_i$

Fig. 2. The basic algorithm to reduce a formula N to its simplified form

Example 6. Consider the following formula: $\underbrace{x \neq 1}_{L_1} \wedge \underbrace{(\underbrace{x \leq 0}_{L_2} \vee \underbrace{x > 2}_{L_3} \vee \underbrace{x = 1}_{L_4})}_{N}$

The simplified form of this formula is $x \leq 0 \vee x > 2$. Assuming we process child L_1 before N in the outer $\wedge$ connective, the critical constraint for L_1 is computed as $x \leq 0 \vee x > 2 \vee x = 1$, which implies neither L_1 nor $\neg L_1$. If we would not resimplify L_1 after simplifying N, the algorithm would (incorrectly) yield $x \neq 1 \wedge (x \leq 0 \vee x > 2)$ as the simplified form of the original formula. However, by resimplifying L_1 after obtaining a simplified $N' = (x \leq 0 \vee x > 2)$, we can now simplify the formula further because the new critical constraint of L_1, $(x \leq 0 \vee x > 2)$, implies $x \neq 1$.

Lemma 6. *The number of validity queries made in the algorithm of Figure 2 is bound by $2n^2$ where n denotes the number of leaves in the initial formula.*

Proof. First, observe that if any call to simplify yields a formula different from the input, the size of this formula must be at least one less than the original formula (see Lemma 3). Furthermore, the number of validity queries made in formula of size k without any simplifications is $2k$. Hence, the total number of validity queries is bound by $2n + 2(n-1) + \ldots + 2$ which is bound by $2n^2$. $\square$

4.2 Making Simplification Practical

In the previous section, we showed that reducing a formula to its simplified form may require making a quadratic number of validity queries. However, these queries are not independent of one another in two important ways: First, all

the formulas that correspond to validity queries share exactly the same set of leaves. Second, the simplification algorithm given in Figure 2 has a push-and-pop structure, which makes it possible to incrementalize queries. In the rest of this section, we discuss how we can make use of these observations to substantially reduce the cost of simplification in practice.

The first observation that all formulas whose satisfiability is queried during the algorithm share the same set of leaves has a fundamental importance when simplifying SMT formulas. Most modern SMT solvers use the DPLL($\mathcal{T}$) framework to solve formulas [17]. In the most basic version of this framework, leaves in a formula are treated as boolean variables, and this boolean overapproximation is then solved by a SAT solver. If the SAT solver generates a satisfying assignment that is not a valid assignment when theory-specific information is accounted for, the theory solver then produces (an ideally minimal) conflict clause that is conjoined with the boolean overapproximation to prevent the SAT solver from generating at least this assignment in the future. Since the formulas solved by the SMT solver during the algorithm presented in Figure 2 share the same set of leaves, theory-specific conflict clauses can be gainfully reused. In practice, this means that after a small number of conflict clauses are learned, the problem of checking the validity of an SMT formula quickly converges to checking the satisfiability of a boolean formula.

The second important observation is that the construction of the critical constraint follows a push-pop stack structure. This is the case because the critical constraint from the parent node is reused, and additional constraints are pushed on the stack (i.e., added to the critical constraint) before the recursive call and (conceptually) popped from the stack after the recursive invocation. This stylized structure is important for making the algorithm practical because almost all modern SAT and SMT solvers support pushing and popping constraints to incrementalize solving. In addition, other tasks that often add overhead, such as CNF construction using Tseitin's encoding for the SAT solver, can also be incrementalized rather than done from scratch. In Section 6, we show the expected overhead of simplifying over solving grows sublinearly in the size of the formula in practice if the optimizations described in this section are used.

5 Integration with Program Analysis

We implemented the proposed algorithm in the Mistral constraint solver [18]. To tightly integrate simplification into a program analysis system, we designed the interface of Mistral such that instead of giving a "yes/no" answer to satisfiability and validity queries, it yields a formula ϕ' in simplified form. Recall that ϕ is satisfiable (valid) if and only if ϕ' is not syntactically *false* (*true*); hence, in addition to obtaining a simplified formula, the program analysis system can check whether the formula is satisfiable by syntactically checking if ϕ' is not *false*. After a satisfiability query is made, we then replace all instances of ϕ with ϕ' such that future formulas that would be constructed by using ϕ are instead constructed using ϕ'. This functionality is implemented efficiently through a

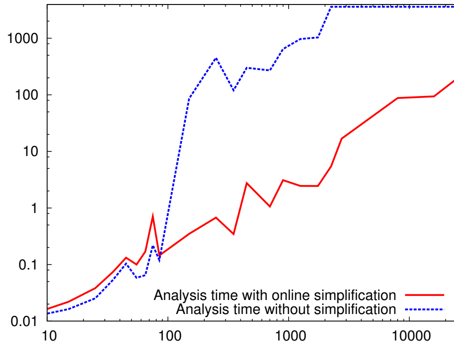


Fig. 3. Running times with and without simplification

shared constraint representation. Hence, Mistral's interface is designed to be useful for program analysis systems that incrementally construct formulas from existing formulas and make many intermediary satisfiability or validity queries. Examples of such systems include, but are not limited to, [10–13, 19, 20].

6 Experimental Results

In this section, we report on our experience using on-line simplification in the context of program analysis. Since the premise of this work is that simplification is useful only if applied continuously during the analysis, we do not evaluate the proposed algorithm on solving off-line benchmarks such as the SMT-LIB. In particular, the proposed technique is not meant as a preprocessing step before solving and is not expected to improve solving time on individual constraints.

6.1 Impact of On-Line Simplification on Analysis Scalability

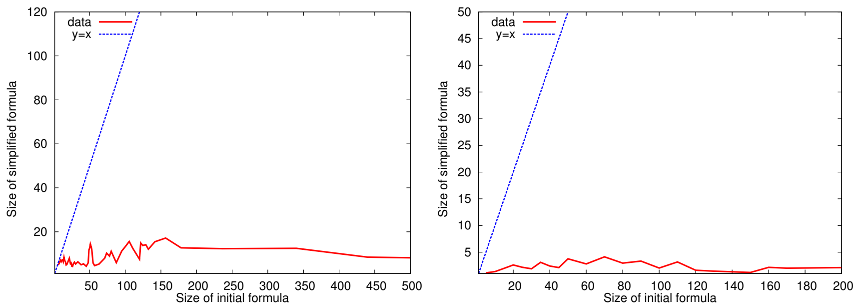
In our first experiment, we integrate Mistral into the Compass program verification system. Compass [19] is a path- and context-sensitive program analysis system for analyzing C programs, integrating reasoning about both arrays and contents of the heap. Compass checks memory safety properties, such as buffer overruns, null dereferences, casting errors, and uninitialized memory; it can also check user-provided assertions. Compass generates constraints in the combined theory of uninterpreted functions and linear integer arithmetic, and as typical of many program analysis systems [13, 19–21], constraints generated by Compass become highly redundant over time, as new constraints are obtained by combining existing constraints. Most importantly, unlike other systems that employ various (usually incomplete) heuristics to control formula size, Compass tracks program conditions precisely without identifying a relevant set of predicates to track. Hence, this experiment is used to illustrate that a program analysis system can be made scalable through on-line simplification instead of using specialized heuristics, such as the ones discussed in Section 1, to control formula size.

In this experiment, we run Compass on 811 program analysis benchmarks, totalling over 173,000 lines of code, ranging from small programs with 20 lines to real-world applications, such as OpenSSH, with over 26,000 lines. For each benchmark, we fix a time-out of 3600 seconds and a maximum memory of 4 GB. Any run exceeding either limit was aborted and assumed to take 3600 seconds.

Figure 3 compares Compass's running times on these benchmarks with and without on-line simplification. The x-axis shows the number of lines of code for various benchmarks and the y-axis shows the running time in seconds. Observe that both axes are log scale. The blue (dotted) line shows the performance of Compass without on-line simplification while the red (solid) line shows the performance of Compass using the simplification algorithm presented in this paper and using the improvements from Section 4.2. In the setting that does not use on-line simplification, Mistral returns the formula unchanged if it is satisfiable and *false* otherwise. As this figure shows, Compass performs dramatically better with on-line simplification on any benchmark exceeding 100 lines. For example, on benchmarks with an average size of 1000 lines, Compass performs about two orders of magnitude better with on-line simplification, and can analyze programs of this size in just a few seconds. Furthermore, using on-line simplification, Compass can analyze benchmarks with a few ten thousand lines of code, such as OpenSSH, in the order of just a few minutes without employing any heuristics to identify relevant conditions.

6.2 Redundancy in Program Analysis Constraints

This dramatic impact of simplification on scalability is best understood by considering how redundant formulas become when on-line simplification is disabled when analyzing the same set of 811 program analysis benchmarks. Figure 4(a) plots the size of the initial formula vs. the size of the simplified formula when formulas generated by Compass are not continuously simplified. The $x = y$ line is plotted as a comparison to show the worst-case when the simplified formula



(a) Size of initial formula vs. size of simplified formula in Compass without simplification

(b) Size of initial formula vs. size of simplified formula in Saturn

Fig. 4. Reduction in the Size of Formulas

is no smaller than the original formula. As this figure shows, while formula sizes grow very quickly without on-line simplification, these formulas are very redundant, and much smaller formulas are obtained by simplifying them. We would like to point out that the redundancies present in these formulas cannot be detected through simple syntactic checks because Mistral still performs extensive syntactic simplifications, such as detecting duplicates, syntactic contradictions and tautologies, and folding constants.

To demonstrate that Compass is not the only program analysis system that generates redundant constraints, we also plot in Figure 4(b) the original formula size vs. simplified formula size on constraints obtained on the same benchmarks by the Saturn program analysis system [13]. First, observe that the constraints generated by Saturn are also extremely redundant. In fact, their average size after simplification is 1.93 whereas the average size before simplification is 73. Second, observe that the average size of simplified constraints obtained from Saturn is smaller than the average simplified formula size obtained from Compass. This difference is explained by two factors: (i) Saturn is significantly less precise than Compass, and (ii) it adopts heuristics to control formula size.

The reader may not find it surprising that the redundant formulas generated by Compass can be dramatically simplified. That is, of course, precisely the point. Compass gains both better precision and simpler engineering from constructing straightforward formulas and then simplifying them because it does not need to heuristically decide in advance which predicates are important. But these experiments also show that the formulas generated by Compass are not unusually redundant to begin with: As the Saturn experiment shows, because analysis systems build formulas compositionally guided by the structure of the program, even highly-engineered systems like Saturn, designed without the assumption of pervasive simplification, can construct very redundant formulas.

6.3 Complexity of Simplification in Practice

In another set of experiments, we evaluate the performance of our simplification algorithm on over 93,000 formulas obtained from our 811 program analysis benchmarks. Recall from Lemma 6 that simplification may require a quadratic number of validity checks. Since the size of the formulas whose validity is checked by the algorithm is at most as large as the original formula, the ratio of simplifying to solving could, in the worst case, be quadratic in the size of the original formula. Fortunately, with the improvements discussed in Section 4.2, we show empirically that simplification adds sub-linear overhead over solving in practice.

Figure 5 shows a detailed evaluation of the performance of the simplification algorithm. In all of these graphs, we plot the ratio of simplifying time to solving time vs. size of the constraints. In graphs 5a and 5c, the constraints we simplify are obtained from analysis runs where on-line simplification is enabled. For the data in graphs 5b and 5d, we disable on-line simplification during the analysis, allowing the constraints generated by the analysis to become much larger. We then collect all of these constraints and run the simplification algorithm on these much larger constraints in order to demonstrate that the simplification

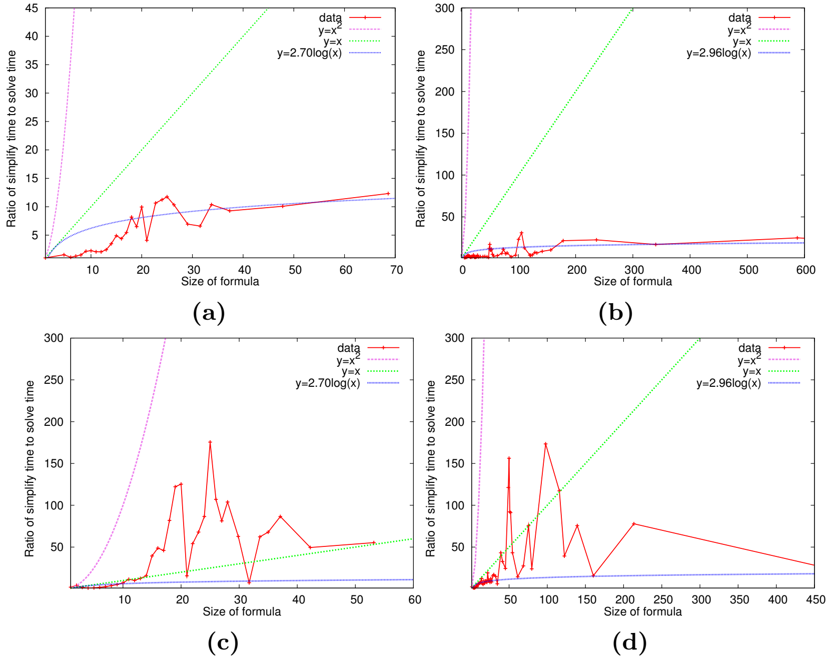


Fig. 5. Complexity of Simplification in Practice

algorithm also performs well on larger constraints with several hundred leaves. In all of these graphs, the red (solid) line marks data points, the blue (lower dotted) line marks the function best fitting the data, the green (middle dotted) line marks $y = x$, and the pink (upper dotted) line marks $y = x^2$. The top two graphs are obtained from runs that employ the improvements described in Section 4.2 whereas the two bottom graphs are obtained from runs that do not. Observe that in graphs 5a and 5b, the average ratio of simplification to solve time seems to grow sublinearly in formula size. In fact, from among the family of formulas $y = cx^2$, $y = cx$, and $y = c \cdot log(x)$, the data in figures 4a and 4b are best approximated by $y = 2.70 \cdot log(x)$ and $y = 2.96 \cdot log(x)$ with asymptotic standard errors 1.98% and 2.42% respectively. On the other hand, runs that do not exploit the dependence between different implication queries exhibit much worse performance, often exceeding the $y = x$ line. These experiments show the importance of exploiting the interdependence between different implication queries and validate our hypothesis that simplifying SMT formulas converges quickly to simplifying SAT formulas when queries are incrementalized. These experiments also show that the overhead of simplifying vs. solving can be made manageable since the ratio of simplifying to solving seems to grow very slowly in the size of the formula.

7 Related Work

Finding simpler representations of boolean circuits is a well-studied problem in logic synthesis and automatic test pattern generation (ATPG) [2, 22, 23]. Our definition of redundancy is reminiscent of the concept of *undetectable faults* in circuits, where pulling an input to 0 (false) or 1 (true) is used to identify redundant circuitry. However, in contrast to the definition of size considered in this paper, ATPG and logic synthesis techniques are concerned with minimizing DAG size, representing the size of the circuit implementing a formula. As a result, the notion of redundancy considered in this paper is different from the notion of redundancy addressed by these techniques. In particular, in our setting, one subpart of the formula may be redundant while another syntactically identical subpart may not. In this paper, we consider different definitions of size and redundancy because except for a few operations like substitution, most operations performed on constraints in a program analysis system are sensitive to the "tree size" of the formula, although these formulas are represented as DAGs internally. Therefore, formulas we consider do not exhibit reconvergent fanout and every leaf has exactly one path from the root of the formula. This observation makes it possible to formulate an algorithm based on critical constraints for simplifying formulas in an arbitrary theory. Furthermore, we apply this simplification technique to on-line constraint simplification in program analysis.

The algorithm we present for converting formulas to simplified form can be understood as an instance of a *contextual rewrite system* [15, 16]. In contextual rewriting systems, if a precondition, called a *context*, is satisfied, a rewrite rule may be applied. In our algorithm, the critical constraint can be seen as a context that triggers a rewrite rule $L \rightarrow true$ if L is implied by the critical constraint α, and $L \rightarrow false$ if α implies $\neg L$. While contextual rewriting systems have been used for simplifying constraints within the solver [16], our goal is to generate an *equivalent* (rather than equisatisfiable) formula that is in simplified form. Furthermore, we propose simplification as an alternative to heuristic-based predicate selection techniques used for improving scalability of program analysis systems.

Finding redundancies in formulas has also been studied in the form of *vacuity detection* in temporal logic formulas [24, 25]. Here, the goal is to identify vacuously valid subparts of formulas, indicating, for example, a specification error. In contrast, our focus is giving a practical algorithm for on-line simplification of program analysis constraints.

The problem of representing formulas compactly has received attention from many different angles. For example, BDDs attempt to represent propositional formulas concisely, but they suffer from the variable ordering problem and are prone to a worst-case exponential blow-up [26]. BDDs have also been extended to other theories, such as linear arithmetic [27–29]. In contrast to these approaches, a formula in simplified form is never larger than the original formula. Loveland and Shostak address the problem of finding a minimal representation of formulas in normal form [30]; in contrast, our approach does not require formulas to be converted to DNF or CNF.

Various rewrite-based simplification rules have also been successfully applied as a preprocessing step for solving, usually for bit-vector arithmetic [31, 32]. These rewrite rules are syntactic and theory-specific; furthermore, they typically yield equisatisfiable rather than equivalent formulas and give no goodness guarantees. In contrast, the technique described in this paper is not meant as a preprocessing step for solving and guarantees non-redundancy.

The importance of on-line simplification of program analysis constraints has been studied previously in the very different setting of set constraints [21]. Simplification based on syntactic rewrite-rules has also been shown to improve the performance of a program analysis system significantly in [33].

Finding redundancies in constraints has also been used for optimization of code in the context of constraint logic programming (CLP) [34]. In this setting, constraint simplification is used for improving the running time of constraint logic programs; however, the simplification techniques considered there do not work on arbitrary SMT formulas.

Acknowledgments

We thank David Dill for his valuable feedback on a draft of this paper and the anonymous reviewers for their detailed and useful comments.

References

1. Een, N., Sorensson, N.: MiniSat: A SAT solver with conflict-clause minimization. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, Springer, Heidelberg (2005)
2. Kim, J., Silva, J., Savoj, H., Sakallah, K.: RID-GRASP: Redundancy identification and removal using GRASP. In: International Workshop on Logic Synthesis (1997)
3. Malik, S., Zhao, Y., Madigan, C., Zhang, L., Moskewicz, M.: Chaff: Engineering an Efficient SAT Solver. In: DAC, pp. 530–535. ACM, New York (2001)
4. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Dutertre, B., De Moura, L.: The Yices SMT Solver. Technical report, SRI (2006)
6. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
7. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
8. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodrıguez-Carbonell, E., Rubio, A.: The Barcelogic SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, p. 294. Springer, Heidelberg (2008)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. JACM 50(5), 752–794 (2003)
10. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM, New York (2002)
11. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: POPL, NY, USA, pp.1–3 (2002)
12. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. ACM SIGPLAN Notices 37(5), 57–68 (2002)

13. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: POPL, vol. 40, pp. 351–363. ACM, New York (2005)
14. Bugrara, S., Aiken, A.: Verifying the safety of user pointer dereferences. In: IEEE Symposium on Security and Privacy, SP 2008, pp. 325–338 (2008)
15. Lucas, S.: Fundamentals of Contex-Sensitive Rewriting. LNCS, pp. 405–412. Springer, Heidelberg (1995)
16. Armando, A., Ranise, S.: Constraint contextual rewriting. Journal of Symbolic Computation 36(1), 193–216 (2003)
17. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). Journal of the ACM (JACM) 53(6), 977 (2006)
18. Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 233–247. Springer, Heidelberg (2009)
19. Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
20. Babić, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: ICSE, pp. 211–220. ACM, New York (May 2008)
21. Faehndrich, M., Foster, J., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: PLDI, p. 96. ACM, New York (1998)
22. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In: DAC, pp.532–535 (2006)
23. Mishchenko, A., Brayton, R., Jiang, J., Jang, S.: SAT-based logic optimization and resynthesis. In: Proc. IWLS 2007, pp. 358–364 (2007)
24. Kupferman, O., Vardi, M.: Vacuity detection in temporal model checking. International Journal on Software Tools for Technology Transfer 4(2), 224–233 (2003)
25. Armoni, R., Fix, L., Flaisher, A., Grumberg, O., Piterman, N., Tiemeyer, A., Vardi, M.: Enhanced vacuity detection in linear temporal logic. LNCS, pp. 368–380. Springer, Heidelberg (2003)
26. Bryant, R.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) 24(3), 293–318 (1992)
27. Bryant, R., Chen, Y.: Verification of arithmetic functions with BMDs (1994)
28. Clarke, E., Fujita, M., Zhao, X.: Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In: ICCAD (1995)
29. Cheng, K., Yap, R.: Constrained decision diagrams. In: Proceedings of the National Conference on Artificial Intelligence, vol. 20, p. 366 (2005)
30. Loveland, D., Shostak, R.: Simplifying interpreted formulas. In: Proc. 5th Conf. on Automated Deduction (CADE), vol. 87, pp. 97–109. Springer, Heidelberg (1987)
31. Ganesh, V., Dill, D.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, p. 519. Springer, Heidelberg (2007)
32. Jha, S., Limaye, R., Seshia, S.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 668–674. Springer, Heidelberg (2009)
33. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. SIGPLAN Not. 44(6), 363–374 (2009)
34. Kelly, A., Marriott, A., Stuckey, P., Yap, R.: Effectiveness of Optimizing Compilation for CLP (R). In: Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, p. 37. The MIT Press, Cambridge (1996)

Compositional Bitvector Analysis for Concurrent Programs with Nested Locks*

Azadeh Farzan and Zachary Kincaid

University of Toronto

Abstract. We propose a new technique to perform bitvector data flow analysis for concurrent programs. Our algorithm works for concurrent programs with nested locking synchronization. We show that this algorithm computes precise solutions (meet over all paths) to bitvector problems. Moreover, this algorithm is compositional: it first solves a local (sequential) data flow problem, and then efficiently combines these solutions leveraging reachability results on nested locks [6,7]. We have implemented our algorithm on top of an existing sequential data flow analysis tool, and demonstrate that the technique performs and scales well.

1 Introduction

Writing concurrent software is difficult and error prone. In principle, static analysis offers an appealing way to mitigate this situation, but dealing with concurrency remains a serious obstacle. Theory and practice of automatically and statically determining dynamic behaviours of concurrent programs lag far behind those for sequential programs. Enumerating all possible interleavings to perform flow-sensitive analyses is infeasible. It is imperative to formulate compositional analysis techniques and proper behaviour abstractions to tame this so-called *interleaving explosion* problem. We believe that the work presented in this paper is a big step in this direction. We propose a compositional algorithm to compute *precise* solutions for bitvector problems for a general and useful class of concurrent programs.

Data flow analysis has proven to be a useful tool for debugging, maintaining, verifying, optimizing, and testing sequential software. Bitvector analyses (also known as the class of gen/kill problems) are a very useful subclass of data flow analyses. Bitvector analyses have been very widely used in compiler optimization. There are a number of applications for precise concurrent bitvector analyses. To mention a few, reaching definitions analysis can be used for precise slicing of concurrent programs with locks, which can be used as a debugging aid for concurrent programs[1]. Both problems of race and atomicity violation detection can be formulated as variations of the reaching definitions analysis. Lighter

* See [5] for an extended version of this paper including proofs and further discussions.

[1] Concurrent program slicing has been discussed previously [11], but to our knowledge there is no method up until now that handles locks precisely.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 253–270, 2010.
© Springer-Verlag Berlin Heidelberg 2010

versions of information flow analyses may also be formulated as bitvector analyses. Precision will substantially decrease the number false positives reported by any of the above analyses.

There is an apparent lack of techniques to precisely and efficiently solve data flow problems, and more specifically bitvector problems for concurrent programs with dynamic synchronization primitives such as locks. The source of this difficulty lies in the lack of a precise and efficient way to represent program paths. Control flow graphs (CFG) are used to represent program paths for most static analyses on *sequential* programs, but concurrent analogs to CFGs suffer major disadvantages. Concurrent adaptations of CFGs mainly fall into two categories: (1) Those obtained by taking the Cartesian product of CFGs for individual threads, and removing inconsistent nodes. These product CFGs are far too large (possibly even infinite) to be practical. (2) Those obtained by taking the union of the CFGs for individual threads, adding inter-thread edges, and performing a may-happen-in-parallel heuristic to get rid of infeasible paths. These union CFGs may still have an abundance of infeasible paths and cannot be used for precise analyses.

Bitvector problems have the interesting property that solving them precisely is possible without analyzing whole program paths. The key observation is that, in a *forward may* bitvector analysis, a fact f is true at a control location c iff there exists a path to c on which f is generated and not subsequently killed; what happens "before" f is generated is irrelevant. Therefore, bitvector problems only require reasoning about *partial* paths starting at a generating transition. For programs with *only* static synchronization (such co-begin/co-end), bitvector problems can be solved with a combination of sequential reasoning and a light concurrent predecessor analysis [9]. Under the concurrent program model in [9], a fact f holds at a control location c if and only if the control location c' at which f is *generated* is an *immediate* concurrent predecessor of c. Therefore, it is sufficient to only consider concurrent paths of length *two* to compute the precise bitvector solution. Moreover, the concurrent predecessor analysis is very simple for co-begin/coend synchronization.

Dynamic synchronization (which was not handled in [9]) reduces the number of feasible concurrent paths in a program, but unfortunately makes their *finite* representation more complex. This complicates data flow analyses, since a precise concurrent data flow analysis must compute the *meet-over-all-feasible-paths* solution, and the analysis should only consider *feasible* paths (that are no longer limited to paths of length two). Evidence of the degree of difficulty that dynamic synchronization introduces is the fact that pairwise reachability (which can be formulated as a bitvector problem) is undecidable for recursive programs with locks. It is however decidable [7] if the locks are acquired in a *nested* manner (i.e. locks are released in the reverse order that they were acquired). We use this result to introduce *sound and complete* abstractions for the set of feasible concurrent paths, which are then used to compute the meet-over-all-feasible-paths solution to the class of bitvector analyses.

We propose a *compositional* (and therefore scalable) technique to *precisely* solve bitvector analysis problems for concurrent programs with nested locks. The analysis proceeds in three phases. In the first phase, we perform the sequential bitvector analysis for each thread individually. In the second phase, we use a sequential data flow analysis to compute an abstract semantics for each thread based on an abstract interpretation of sequential trace semantics. We then combine the abstract semantics for each pair of threads to compute a second set of data flow facts, namely those who reach concurrently. In the third phase, we simply combine the results of the sequential and concurrent phases into a *sound and complete* final solution for the problem. This procedure is quadratic in the number of threads and exponential (in the worst case) in the number of shared locks in the program; however, we do not expect to encounter even close to the worst case in practice. In fact, in our experiments the running time follows a growth pattern that almost matches that of sequential programs. Our approach avoids the limitations typically imposed by concurrent adaptations of CFGs: it is scalable and compositional, in contrast with the product CFG; and it is precise, in contrast with union CFGs.

In this paper we discuss the class of *intraprocedural forward may* bitvector analyses for a concurrent program model with nested locking as our main contribution. Nested locks are a common programming practice; for example Java synchronized methods and blocks syntactically enforce this condition. Due to lack of space, all further discussions on the generalization of this case have been included in an extended version of this paper, which includes discussions on backward and interprocedural analyses, as well as an extension to a parameterized concurrent program model.

We have implemented our algorithm on top of the C language front-end CIL [18], which performs the sequential data flow analyses required by our algorithm. We show through experimentation that this technique scales well and has running time close to that of sequential analysis in practice.

Related Work. Program flow analysis was originally developed for sequential programs to enable compiler optimizations [1]. Although the majority of flow analysis research has been focused on sequential software [19,15,20], flow analysis for concurrent software has also been studied. Flow-insensitive analyses can be directly adapted into the concurrent setting. Existing flow-sensitive analyses [14,16,17,21] have at least one of the following two restrictions: (a) the programs they handle have extremely simplistic concurrency/synchronization mechanisms and can be handled precisely using the union of control flow graphs of individual programs, or (b) the analysis is sound but not complete, and solves the data flow problem using heuristic approximations.

RADAR [2] attempts to address some of the problems mentioned above, and achieves scalability and more precision by using a race detection engine to kill the data flow facts generated and propagated by the sequential analysis. RADAR's degree of precision and performance depends on how well the race detection engine works. We believe that although RADAR is a good practical solution,

it does not attempt to solve the real problem at hand, nor does it provide any insights for static analysis of concurrent programs.

Knoop et al [9] present a bitvector analysis framework which comes closest to ours in that it can express a variety of data flow analysis problems, and gives sound and complete algorithms for solving them. However, it cannot handle dynamic synchronization mechanisms (such as locks). This approach has been extended for the same restricted synchronization mechanism to handle procedures in [3,4,22] and generalizations of bitvector problems in [10,22].

Foundational work on nested locks appears in [6,7]. Recently, analyses based on this work have been developed, including [8] and [12]. Notably, the authors of [8] detect violations of properties that can be expressed as phase automata, which is a more general problem than bitvector analysis. However, their method is not tailored to bitvector analysis, and is not practically viable when a "full" solution (a solution for every fact and every control location) to the problem is required, which is often the case.

2 Preliminaries

A concurrent program $\mathcal{CP}$ is a pair $(\mathcal{T}, \mathcal{L})$ consisting of a finite set of threads $\mathcal{T}$ and a finite set of locks $\mathcal{L}$. We represent each thread $T \in \mathcal{T}$ as a control flow automaton (CFA). CFAs are similar to a control flow graphs, except actions are associated with edges (which we will call transitions) rather than nodes. Formally, a CFA is a graph (N_T, Σ_T) with a unique entry node s_T and a function $stmt_T : \Sigma_T \to Stmt$ that maps transitions to program statements. We assume no two threads have a common node (transition), and refer to the set of all nodes (transitions) by N (Σ). In the following, we will often identify transitions with their corresponding program statements. CFA statements execute atomically, so in practice we split non-atomic statements prior to CFA construction.

For each lock $l \in \mathcal{L}$, we distinguish two synchronization statements $acq(l)$ and $rel(l)$ that acquire and release the lock l, respectively. Locks are the only means of synchronization in our concurrent program model. For a finite path π through thread T starting at s_T, we let $\text{Lock-Set}_T(\pi)$ denote the set of locks held by T after executing π[2].

A local run of thread T is any finite path starting at its entry node; we refer to the set of all such runs by $\mathcal{R}_T$. A run of $\mathcal{CP}$ is a sequence $\rho = t_1 \dots t_n \in \Sigma^\star$ of transitions such that:

i) ρ projected onto each thread T (denoted by ρ_T), is a local run of T
ii) There exists no point p along ρ at which two threads T, T' hold the same lock ($\nexists T, T', p. T \neq T' \wedge \text{Lock-Set}_T((t_1 \cdots t_p)_T) \cap \text{Lock-Set}_{T'}((t_1 \cdots t_p)_{T'}) \neq \emptyset$).

We use $\mathcal{R}_{\mathcal{CP}}$ to denote the set of all runs of $\mathcal{CP}$ (just $\mathcal{R}$ when there is no confusion about $\mathcal{CP}$). For a sequence $\rho = t_1 \dots t_n \in \Sigma^\star$ and $1 \leq r \leq s \leq n$ we use $\rho[r]$ to denote t_r, $\rho[r, s]$ to denote $t_r \dots t_s$, and $|\rho|$ to denote n.

[2] Formally, $\text{Lock-Set}_T(\pi) = \{l \in \mathcal{L} \mid \exists i. \pi_T[i] = acq(l) \wedge \nexists j > i \ s.t. \ \pi_T[j] = rel(l)\}$.

A program $\mathcal{CP}$ *respects nested locking* if for every thread $T \in \mathcal{T}$ and for every local run π of T, π releases locks in the opposite order it acquires them. That is, there exists no l, l' such that π contains a contiguous subsequence $acq(l); acq(l'); rel(l)$ when projected onto the the acquire and release transitions of l and l'[3].

From this point on, whenever we refer to a concurrent program $\mathcal{CP}$, we assume that it respects nested locking. Restricting our attention to programs that respect nested locking allows us to keep reasoning about run interleavings tractable. We will make critical use of this assumption in the following.

2.1 Locking Information

Consider the example in Figure 1. We would like to know whether the fact d generated at the location b reaches the location a (without being killed at location c). If the thread on the right takes the `else` branch in the first execution of the loop, it will have to go through location c and kill the fact d before the execution of the program can get to location a. However, if the program takes the `then` branch in the first iteration of the loop and takes the `else` branch in the second one, then execution can follow to a without having to kill d first. This example shows that in general, the sorts of interleavings that we must consider in a bitvector analysis can be quite complicated.

```
acq(l2);
   acq(l1);
      ...
   rel(l1);
   a: ...
rel(l2);
```

```
acq(l1);
   acq(l2);
      ...
   rel(l2);
   while (...) {
      if (...) {
         rel(l1);
         acq(l1);
      } else {
         b: ... // gen "d"
      }
   }
   c: ... // kill "d"
rel(l1);
```

Fig. 1. Locking information

In [6] and [7], compositional reasoning approaches for programs that respect nested locking were introduced, which are based on *local* locking information. We quickly give an overview of this here. In the following, $T \in \mathcal{T}$ denotes a thread, and $\rho \in \Sigma_T^\star$ denotes a sequence of transitions of T (in practice, ρ will be a run or a suffix of a run of T).

- Locks-Held$_T(\rho, i) = \{l \in \mathcal{L} \mid \forall k \geq i. l \in \text{Lock-Set}_T(\rho[1,k])\}$: the set of locks held continuously by T through ρ, starting no later than at position i.
- Locks-Acq$_T(\rho) = \{l \in \mathcal{L} \mid \exists k. \rho[k] = T{:}acq(l)\}$: the set of locks that are acquired by T along ρ.
- $fah_T(\rho)$ (forward acquisition history): a partial function which maps each lock l whose last acquire in ρ has no matching release, to the set of locks that were acquired after the last acquisition of l (and is undefined otherwise)[4].
- $bah_T(\rho, i)$ (backward acquisition history): a partial function which maps each lock l that is held at $\rho[i]$ and is released in $\rho[i, |\rho|]$ to the set of locks that were released before the first release of l in $\rho[i, |\rho|]$ (and is undefined otherwise).

[3] In the special case where $l = l'$, this condition implies that locks are not re-entrant.

[4] Note that the domain of $fah_T(\rho)$ (denoted $dom(fah_T(\rho))$) is exactly Lock-Set$_T(\rho)$.

We omit T subscripts for all of these functions when T is clear from the context.

As observed in [6,7], a necessary and sufficient condition for pairwise reachability cannot be stated in terms of locksets (the "current" lock behaviour of each thread). One needs to additionally consider the historical lock behaviour (fah and bah) of each thread, which places ordering constraints on locking events. This notion will be made more precise in Proposition 2; see [6,7] for more details. As an example of fah and bah, consider Figure 1. The run of the right thread that starts at the beginning, enters the `while` loop, and takes the `else` branch to end at `b` has forward acquisition history $[\texttt{l1} \mapsto \{\texttt{l2}\}]$. If that run continues to loop, taking the `then` branch and then the `else` branch to end back at `b`, that run has forwards acquisition history $[\texttt{l1} \mapsto \{\}]$. The run of the left thread that executes the entire code block has backwards acquisition history $[\texttt{l2} \mapsto \{\}]$ at `a` and $[\texttt{l2} \mapsto \{\texttt{l1}\}; l1 \mapsto \{\}]$ between the acquire and release of `l1`.

2.2 Bitvector Data Flow Analysis

Let $\mathbb{D}$ be a finite set of data flow facts of interest. The goal of data flow analysis is to replace the *full* semantics by an abstract version which is tailored to deal with a specific problem. The abstract semantics is specified by a local semantic functional $[\![\cdot]\!]_{\mathbb{D}} : \Sigma \to (\wp(\mathbb{D}) \to \wp(\mathbb{D}))$ where for each transition t, $[\![t]\!]_{\mathbb{D}}$ denotes the transfer function associated with t. $[\![\cdot]\!]_{\mathbb{D}}$ gives abstract meaning to every CFA transition (program statement) in terms of a transformation function from a semi-lattice $(\wp(\mathbb{D}), \sqcap)$ (where $\sqcap$ is $\cup$ or $\cap$) into itself. We will drop $\mathbb{D}$ and simply use $[\![t]\!]$ when $\mathbb{D}$ is clear from the context. We extend $[\![\cdot]\!]$ from transitions to transition sequences in the natural way: $[\![\epsilon]\!] = id$, and $[\![t\rho]\!] = [\![\rho]\!] \circ [\![t]\!]$.

Bitvector problems can be characterized by the simplicity of their local semantic functional $[\![\cdot]\!]$: for any transition t, there exist sets $gen(t)$ and $kill(t)$ ($\subseteq \mathbb{D}$) such that $[\![t]\!](D) = (D \cup gen(t)) \setminus kill(t)$. Equivalently, for any t, $[\![t]\!]$ can decomposed into $|\mathbb{D}|$ monotone functions $[\![t]\!]_i : \mathbb{B} \to \mathbb{B}$, where $\mathbb{B}$ is the Boolean lattice $(\{\mathrm{ff}, \mathrm{tt}\}, \Rightarrow)$.

Our goal is to compute the concurrent meet-over-paths ($CMOP$) value of transition[5] t of $\mathcal{CP}$, defined as

$$CMOP[t] = \bigsqcap_{\rho t \in \mathcal{R}_{\mathcal{CP}}} [\![\rho]\!]_{\mathbb{D}}(\top_{\mathbb{D}})$$

$CMOP[t]$ is the optimal solution to the data flow problem. Note in particular that only runs that respect the semantics of locking contribute to the solution. This definition is not effective, however, since $\mathcal{R}_{\mathcal{CP}}$ may be infinite; the contribution of this work is an efficient algorithm for computing $CMOP[t]$.

[5] For the CFA formulation of data flow analysis, data flow transformation functions and solutions correspond to transitions rather than nodes.

3 Concurrent Data Flow Framework

Fix a concurrent program $\mathcal{CP}$ with set of threads $\mathcal{T}$, set of locks $\mathcal{L}$, and a set of data flow facts $\mathbb{D}$ with meet $\cup$ (bitvector problems that use $\cap$ for meet can be solved using their dual problem). For a data flow fact $d \in \mathbb{D}$, and for a transition t, let $[\![t]\!]_d$ denote $[\![t]\!]_{\mathbb{D}}$ projected onto d (defined by $[\![t]\!]_d(p) = (p \vee d \in gen(t)) \wedge d \notin kill(t)$). Call a sequence π **d-preserving** if $[\![\pi]\!]_d = id$. In particular, the empty sequence ϵ is d-preserving for any $d \in \mathbb{D}$.

The following observation from [9] is the key to the efficient computation of the *interleaving effect*. It pinpoints the specific nature of a semantic functional for bitvector analysis, whose codomain only consists of *constant functions* and the *identity*:

Lemma 1. *[9] For a data flow fact $d \in \mathbb{D}$, and a transition t of a concurrent program $\mathcal{CP}$, $d \in CMOP[t]$ iff there exists a run $t_1 \cdots t_n t \in \mathcal{R}_{\mathcal{CP}}$ and there exists k, $(1 \leq k \leq n)$ such that $[\![t_k]\!]_d = const_{tt}$ and for all m, $(k < m \leq n)$, we have $[\![t_m]\!]_d = id$.*

Call such a run a **d-generating run** for t, and call t_k the generating transition of that run.

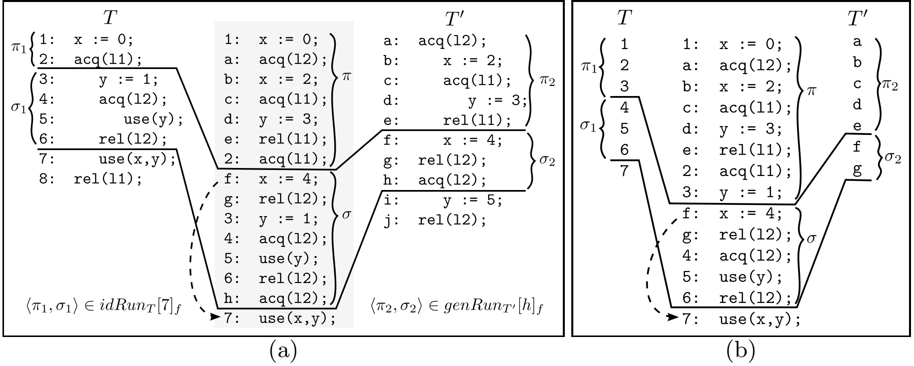


Fig. 2. A witness run (a) and a *normal* witness run (b) for definition `f` reaching `7`

This lemma restricts the possible interference within a concurrent program: *if there is any interference, then the interference is due to a single statement within a parallel component*. By *interference*, we mean any possible behaviour by other threads that may change the set of facts that hold in a program location; in the realm of the gen/kill problems, this may be in the form of a fact that sequentially holds getting killed, or a fact that does not sequentially hold getting generated. Consider the program in Figure 2(a). In a reaching definitions analysis, only transition `f` (of T') can generate the "definition at `f` reaches" fact. For any witness trace and any fact d, we can pinpoint a single transition that generates this fact (namely, the last occurrence of a generating transition on that trace).

This is *not* true for data flow analyses which are not bitvector analyses. For example, in a null pointer dereference analysis, witnesses may contain a chain of assignments, no single one of which is "responsible" for the pointer in question being null, but *combined* they make the value of a pointer null. Our algorithm critically takes advantage of the simplicity of bitvector problems to achieve both efficiency and precision, and cannot be trivially generalized to handle all data flow problems.

Based on Lemma 1 and the observation from [7] that runs can be projected onto runs with fewer threads, we get the following:

Lemma 2. *For a data flow fact $d \in \mathbb{D}$, and for a transition t of thread T, there exists a d-generating run for t if and only if one of the following holds:*

- *There exists a local d-generating run for t (that is, a d-generating run consisting only of transitions from T). Call such a run a* single-indexed d-generating run.
- *There exists a thread T' ($T \neq T'$) such that there is a d-generating run π for t consisting only of transitions from T and T' and such that the generating transition of π belongs to T'. Call such a run a* double-indexed d-generating run.

Thus, to determine whether $d \in CMOP[t]$ (i.e. fact d may be true at t), it is sufficient to check whether there is a single- or double-indexed d-generating run to t. Therefore, the precise solution to the concurrent bitvector analysis problem can be computed by only reasoning about concurrent programs with one or two threads, so long as we consider each pair of threads in the system. The existence of a single-indexed d-generating run to t can be determined by a *sequential* bitvector data flow analysis, which have been studied extensively.

Here, we discuss a compositional technique for enumerating the double-indexed d-generating runs. In order to achieve compositionality, we (1) characterize double-indexed d-generating runs in terms of two local runs, and (2) provide a procedure to determine whether two local runs can be combined into a global run. First, we define for each thread T, each transition t of T, and each data flow fact $d \in \mathbb{D}$:

- $PR_T[t]_d = \{\langle \pi, \sigma \rangle \mid \pi\sigma t \in \mathcal{R}_T \ \wedge [\![\sigma]\!] = id\}$
- $GR_T[t]_d = \{\langle \pi, \sigma \rangle \mid \ \pi\sigma \in \mathcal{R}_T \ \wedge \ [\![\sigma[1]]\!] = const_{tt} \ \wedge \ [\![\sigma[2, |\sigma| - 1]]\!] = id \ \wedge \sigma[|\sigma|] = t\}$

Intuitively, $PR_T[t]_d$ and $GR_{T'}[t']_d$ correspond to sets of *local* runs of threads T and T' which can be combined (interleaved in a lock-valid way) to create a *global* d-generating run to t. For example, in Figure 2(a), the definition at line `f` reaches the use at line `7` (in a reaching-definitions analysis) since the local runs $\pi_1\sigma_1$ of T and $\pi_2\sigma_2$ of T' can be combined into the run $\pi\sigma$ (demonstrated in the center) to create a double-indexed generating run. The following proposition is the key to our compositional approach:

Proposition 1. *Assume a concurrent program* $\mathcal{CP}$ *with two threads* T_1 *and* T_2*. There exists a double-indexed d-generating run to transition* t_1 *of thread* T_1 *if and only if there exists a transition* t_2 *of thread* T_2 *such that there exists* $\langle \pi_1, \sigma_1 \rangle \in PR_{T_1}[t_1]_d$ *and* $\langle \pi_2, \sigma_2 \rangle \in GR_{T_2}[t_2]_d$ *and a run* $\pi\sigma \in \mathcal{R}_{\mathcal{CP}}$ *such that* $\pi_{T_1} = \pi_1$*,* $\pi_{T_2} = \pi_2$*,* $\sigma_{T_1} = \sigma_1$ *and* $\sigma_{T_2} = \sigma_2$*.*

Since PR and GR are sets of *local* runs, they can be computed locally and independently, and checked whether they can be interleaved in a second phase. However, $PR_T[t]_d$ and $GR_T[t]_d$ are (in the general case) infinite sets, so we need to find finite means to represent them. In fact, we do not need to know about all such runs: the only thing that we need to know is whether there exists *a* d-generating run in one thread, and *a* d-preserving run in the other thread that *can be combined* into a lock-valid run to carry the fact d generated in one thread to a particular control location in the other thread. Proposition 2, a simple consequence of a theorem from [6], provides a means to represent these sets with finite abstractions.

Proposition 2. *Let* $\mathcal{CP}$ *be a concurrent program, and let* T_1*,* T_2 *be threads of* $\mathcal{CP}$*. Let* $\pi_1\sigma_1$ *be a local run of* T_1 *and let* $\pi_2\sigma_2$ *be a local run of* T_2*. Then there exists a run* $\pi\sigma \in \mathcal{R}_{\mathcal{CP}}$ *with* $\pi_{T_1} = \pi_1$*,* $\pi_{T_2} = \pi_2$*,* $\sigma_{T_1} = \sigma_1$*, and* $\sigma_{T_2} = \sigma_2$ *if and only if:*

- $\mathit{Lock\text{-}Set}(\pi_1) \cap \mathit{Lock\text{-}Set}(\pi_2) = \emptyset$
- $fah(\pi_1)$ *and* $fah(\pi_2)$ *are consistent*[6]
- $\mathit{Lock\text{-}Set}(\pi_1\sigma_1) \cap \mathit{Lock\text{-}Set}(\pi_2\sigma_2) = \emptyset$*.*
- $fah(\pi_1\sigma_1)$ *and* $fah(\pi_1\sigma_2)$ *are consistent*
- $bah(\pi_1\sigma_1, |\pi_1|)$ *and* $bah(\pi_2\sigma_2, |\pi_2|)$ *are consistent*
- $\mathit{Locks\text{-}Acq}(\sigma_1) \cap \mathit{Locks\text{-}Held}(\pi_2\sigma_2, |\pi_2|) = \emptyset$ *and* $\mathit{Locks\text{-}Acq}(\sigma_2) \cap \mathit{Locks\text{-}Held}(\pi_1\sigma_1, |\pi_1|) = \emptyset$*.*

Observe that Proposition 2 states that one can check whether two *local* runs can be interleaved into a *global* run by performing a few consistency checks on finite representations of the local lock behaviour of the two runs. In other words, one does not have to know what the runs are; one has to only know what the locking information for the runs are. Therefore, we use this information as our finite representation for the set of runs; more precisely, we use a quadruple consisting of two forwards acquisition histories, a backwards acquisition history, and a set of locks acquired to represent an *abstract* run[7]. Let $\mathcal{P}$ be the set of all such abstract runs. We say that two run abstractions are *compatible* if they may be interleaved (according to Proposition 2). We then define an abstraction function $\alpha : \Sigma^\star \times \Sigma^\star \to \mathcal{P}$ that computes the abstraction of a run:

$$\alpha(\langle \pi, \sigma \rangle) = \langle fah(\pi), fah(\pi\sigma), bah(\pi\sigma, |\pi|), \text{Locks-Acq}(\sigma) \rangle$$

[6] Histories h and h' are consistent iff $\nexists \ell \in dom(h), \ell' \in dom(h').\ell \in h'(\ell') \wedge \ell' \in h(\ell)$.

[7] Note that for a run $\pi\sigma$, we can compute Lock-Set(π) as $dom(fah(\pi))$, Lock-Set($\pi\sigma$) as $dom(fah(\pi\sigma))$, and Locks-Held($\pi\sigma, |\pi|$) as $(dom(fah(\pi)) \cap dom(fah(\pi\sigma))) \setminus$ Locks-Acq(σ).

For each transition $t \in \Sigma_T$ and data flow fact d, this abstraction function can be applied to the sets $PR_T[t]_d$ and $GR_T[t]_d$ to yield the sets $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$, respectively:

$$\widehat{PR}_T[t]_d = \{\alpha(\langle \pi, \sigma \rangle) \mid \langle \pi, \sigma \rangle \in PR_T[t]_d\}$$
$$\widehat{GR}_T[t]_d = \{\alpha(\langle \pi, \sigma \rangle) \mid \langle \pi, \sigma \rangle \in GR_T[t]_d\}$$

For example, the abstraction of the preserving run $\pi_1\sigma_1$ and the generating run $\pi_2\sigma_2$ from Figure 2(a) are (in order):

$$\langle \underbrace{[l_1 \mapsto \{\}]}_{fah(\pi_1)}, \underbrace{[l_1, \mapsto \{l_2\}]}_{fah(\pi_1\sigma_1)}, \underbrace{[l_2 \mapsto \{\}]}_{bah(\pi_1\sigma_1, |\pi_1|)}, \underbrace{\{l_2\}}_{\text{Locks-Acq}(\sigma_1)} \rangle$$
$$\langle \underbrace{[l_2 \mapsto \{l_1\}]}_{fah(\pi_2)}, \underbrace{[l_2 \mapsto \{\}]}_{fah(\pi_2\sigma_2)}, \underbrace{[l_2 \mapsto \{\}]}_{bah(\pi_2\sigma_2, |\pi_2|)}, \underbrace{\{l_2\}}_{\text{Locks-Acq}(\sigma_2)} \rangle$$

The definitions of $\widehat{PR}$ and $\widehat{GR}$, and Proposition 2 imply the following proposition:

Proposition 3. *Assume a concurrent program $\mathcal{CP}$ with two threads T_1 and T_2. There exists a double-indexed d-generating run to transition t_1 of thread T_1 if and only if there exists a transition t_2 of thread T_2 such that there exists elements of $\widehat{PR}_{T_1}[t_1]_d$ and $\widehat{GR}_{T_2}[t_2]_d$ which are compatible.*

For a fact d and a transition $t \in \Sigma_T$, the sets $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ are finite, and therefore one can use Proposition 3 to provide a solution to the concurrent bitvector problem, once $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ have been computed (we refer the reader to an extended version of this paper [5] for the details of these computations). In the next section, we propose an optimization that provides the same solution using a potentially much smaller subsets of these sets.

3.1 Normal Runs

The sets $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ may be large in practice, so we introduce the concept of *normal runs* to replace $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ with smaller subsets that are still sufficient for solving bitvector problems. Intuitively, normal runs minimize the number of transitions between the generating transition and the end of the (double-indexed) run. Consider the run in Figure 2(a): it is a witness for definition at `f` reaching the use at `7`, but it is not *normal*: Figure 2(b) pictures a witness consisting of the same transitions (except `h` has been removed from the end of σ_2), which has a shorter σ component. Note that runs that are minimal in this sense are indeed normal; the reverse, however, does not hold. We define normal runs formally as follows:

Definition 1. *Call a double-indexed d-generating run $\pi\sigma t$ (consisting of transitions of threads T and T', where t is a transition of T) with generating transition $\sigma[1]$ (of thread T')* normal *if:*

– $|\sigma| = 1$ *(that is,* $\sigma[1]$ *is an immediate predecessor of* t*)*, or
– *All of the following hold:*
  - *The first* T *transition in* σt *is an* acquire *transition.*
  - *The last* T' *transition in* σ *is a* release *transition.*
  - $\nexists i$ *(*$1 \leq i \leq |\sigma|$*) such that after executing* $\pi(\sigma[1, i])$, T *frees all held locks.*
  - $\nexists i$ *(*$1 < i \leq |\sigma|$*) such that after executing* $\pi(\sigma[1, i])$, T' *frees all held locks.*

Note that if there are no locking operations in a run, then the generating transition is *always* an immediate predecessor of the t, since there are no synchronization restriction to prevent this from happening. We show that it is sufficient to consider only *normal* runs for our analysis, by proving that the existence of a double-indexed d-generating run implies the existence of a normal double-index d-generating run.

Lemma 3. *Let* $t_1 \cdots t_n \in \mathcal{R}_T$ *and let* $t'_1 \cdots t'_m \in \mathcal{R}_{T'}$*. If there is a run* $\rho = \pi\sigma$ *(*$\rho \in \mathcal{R}_{\mathcal{CP}}$*) such that there exist* $1 \leq i \leq n$ *and* $1 \leq j \leq m$ *where:*

$$\begin{aligned} \pi_T &= t_1 \ldots t_i & \sigma_T &= t_{i+1} \ldots t_n \\ \pi_{T'} &= t'_1 \ldots t'_j & \sigma_{T'} &= t'_{j+1} \ldots t'_m \end{aligned}$$

Then, the following hold:

1. *If* t'_{j+1} *is not an acquire transition, then* $\exists \pi', \sigma'$ *such that* $\pi'\sigma' \in \mathcal{R}_{\mathcal{CP}}$*, and*

$$\begin{aligned} \pi'_T &= t_1 \ldots t_i & \sigma'_T &= t_{i+1} \ldots t_n \\ \pi'_{T'} &= t'_1 \ldots t'_{j+1} & \sigma'_{T'} &= t'_{j+2} \ldots t'_m \end{aligned}$$

2. *If* t'_m *is not a release, then* $\exists \sigma'$ *such that* $\pi\sigma' \in \mathcal{R}_{\mathcal{CP}}$ *is a valid run, and*

$$\begin{aligned} \pi'_T &= t_1 \ldots t_i & \sigma'_T &= t_{i+1} \ldots t_n \\ \pi'_{T'} &= t'_1 \ldots t'_j & \sigma'_{T'} &= t'_{j+1} \ldots t'_{m-1} \end{aligned}$$

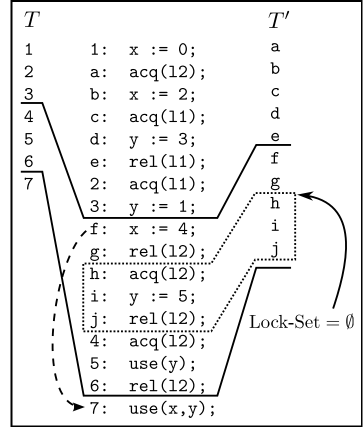
Lemma 3 is a consequence of Lipton's theory of reduction [13]. It is used to trim the beginning of a d-preserving run if it does not start with an *acquire* (part 1) and the end of a d-generating run if it does not end in a *release* (part 2). The run in Figure 2(b) is obtained from the run in Figure 2(a) by an application of Lemma 3.

Lemma 4. *If there is a run* $\pi\sigma$ *of concurrent program* $\mathcal{CP}$ *(consisting of two threads* T *and* T'*) such that* $\pi_T = t_1 \ldots t_i$, $\sigma_T = t_{i+1} \ldots t_n$, $\pi_{T'} = t'_1 \ldots t'_j$ *and* $\sigma_{T'} = t'_{j+1} \ldots t'_m$*, and if there exists* k *(*$j < k \leq m$*) such that Lock-Set*$_{T'}(t'_1 \ldots t'_k) = \emptyset$*, then there exists a run* $\pi'\sigma'$ *of* $\mathcal{CP}$ *where*

$$\begin{aligned} \pi'_T &= t_1 \ldots t_i & \sigma'_T &= t_{i+1} \ldots t_n \\ \pi'_{T'} &= t'_1 \ldots t'_k & \sigma'_{T'} &= t'_{k+1} \ldots t'_m \end{aligned}$$

Similarly, if there exists k *(*$j \leq k \leq m-1$*) such that Lock-Set*$_{T'}(t'_1 \ldots t'_k) = \emptyset$*, then there exists a run* $\pi\sigma'$ *where* $\sigma'_T = \sigma_T$ *and* $\sigma'_{T'} = t_{i+1} \ldots t_k$*.*

Lemma 4 is a consequence of Proposition 2. It is used to trim the beginning of d-preserving runs and the end of d-generating runs. The figure to the right illustrates the application of this lemma: thread T' holds no locks after executing `g`, so transitions `h`, `i`, and `j` need not be executed. The witness pictured for definition `f` reaching `7` corresponds to the normal witness obtained by removing the dotted box.



The following Proposition, which is a consequence of Lemmas 3 and 4, implies that it is sufficient to only consider *normal* runs for the analysis. Therefore, we can ignore runs that are not normal without sacrificing soundness.

Proposition 4. *If there exists a double-indexed d-generating run of concurrent program $\mathcal{CP}$ leading to a transition t, then there exists a normal double-indexed d-generating run of $\mathcal{CP}$ leading to t.*

Therefore, for any transition t and data flow fact d, we define normal versions (subsets of these sets which contain only normal runs) of $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ as follows:

– $\widehat{NPR}_T[t]_d = \{\alpha(\langle\pi,\sigma\rangle) \mid \langle\pi,\sigma\rangle \in PR_T[t]_d \wedge (|\sigma| = 0 \vee (\nexists k.\text{Lock-Set}(\pi(\sigma[1,k])) = \emptyset \wedge \sigma[1]\text{is an acquire}))\}$
– $\widehat{NGR}_T[t]_d = \{\alpha(\langle\pi,\sigma\rangle) \mid \langle\pi,\sigma\rangle \in GR_T[t]_d \wedge(|\sigma| = 1 \vee \nexists k.\text{Lock-Set}(\pi(\sigma[1,k])) = \emptyset)\}$

$\widehat{NPR}_T[t]_d$ ($\widehat{NGR}_T[t]_d$) is a finite representation of sets of normal d-preserving (generating) runs. In order to compute solutions for every data flow fact in $\mathbb{D}$ simultaneously, we extend $\widehat{NPR}_T[t]_d$ and $\widehat{NGR}_T[t]_d$ to *sets* of data flow facts. We define $\widehat{NPR}_T[t]$ to be a partial function that maps each abstract run $\widehat{\rho}$ to the set of facts d for which there is a d-preserving run whose abstraction is $\widehat{\rho}$, and is undefined if there is no concrete run to t whose abstraction is $\widehat{\rho}$. $\widehat{NGR}_T[t]$ is defined analogously. We refer the reader to an extended version of this paper [5] for more detailed information on $\widehat{NPR}$ and $\widehat{NGR}$.

4 The Analysis

Here, we summarize the results presented in previous sections into a procedure for the bitvector analysis of concurrent programs. The procedure is outlined in Algorithm 1. Data flow facts reaching a transition t are computed in two different groups as per Lemma 2: facts with single-indexed generating runs and facts with double-indexed generating runs.

Algorithm 1. Concurrent Bitvector Analysis

1: Compute Summaries and Helper Sets // *see Algorithm 2*
2: **for** each $T \in \mathcal{T}$ **do**
3: Compute MFP_T // *Single-indexed facts*
4: **for** each $t \in \Sigma_T$ **do**
5: Compute $NDIF_T[t]$ // *(Normal) double-indexed facts*
6: $CMOP_T[t] := MFP_T[t] \cup NDIF_T[t]$
7: **end for**
8: **end for**

On line 6, the facts from single- and double-indexed generating runs are combined into the solution of the concurrent bitvector analysis. Facts from single-indexed generating runs can be computed efficiently using well-known (maximum fixed point) sequential data flow analysis techniques. It remains to show how to to efficiently compute facts from double-indexed generating runs using the summaries and helper sets which are computed at the beginning of the analysis.

A naive way to compute $NDIF$ would involve iterating over all pairs of transitions from different threads to find compatible elements of $\widehat{NPR}$ and $\widehat{NGR}$. This would create an $|\Sigma|^2$ factor in our algorithm, which can be quite large. We avoid this by computing thread summaries for each thread T. The summary, $NGen_T$, combines information about each transition in T that is relevant for $NDIF$ computation for other threads. More precisely, $NGen_T$ is a function that maps each run abstraction p to the set of facts for which there is a generating run whose abstraction is p. Intuitively, $NGen_T$ groups together transitions that have similar locking information so that they can be processed at once. This speeds up our analysis significantly in practice.

Algorithm 2. Computing Summaries

1: **for** each $T \in \mathcal{T}$ **do**
2: Compute $\widehat{NGR}_T$ // *Normal generating runs*
3: $NGen_T := \lambda\widehat{\rho}.\bigcup\{\widehat{NGR}_T[t](\widehat{\rho}) \mid t \in \Sigma_T \wedge \widehat{\rho} \in dom(\widehat{NGR}_T[t])\}$
4: Compute $\widehat{NPR}_T$ // *Normal preserving runs*
5: **end for**

The essential procedure for finding facts in $NDIF_T[t]$ is to: 1) find compatible (abstract) runs $\widehat{\rho} \in dom(\widehat{NPR}_T[t])$ and $\widehat{\rho}' \in dom(NGen'_T)$, and 2) add facts that are *both* preserved by $\widehat{\rho}$ and generated by $\widehat{\rho}'$. This procedure is elaborated in Algorithm 3.

It remains to show how to compute $\widehat{NPR}$ and $\widehat{NGR}$. Both of these sets can be computed by a sequential data flow analysis. This analysis is based on an abstract interpretation of sequential trace semantics suggested by the abstraction function α. Best abstract transformers can derived from the definition α; more detailed information can be found in the extended version of this paper.

Algorithm 3. Compute NDIF (concurrent facts from non-predecessors)

Input: Thread T, transition $t \in \Sigma_T$
Output: $NDIF_T[t]$.
1: $NDIF_T[t] := \emptyset$
2: **for** each $T' \neq T$ in $\mathcal{T}$ **do**
3: **for** each $\widehat{\rho} \in dom(\widehat{NPR}_T[t])$ **do**
4: $NDIF_T[t] := NDIF_T[t] \cup \{NGen_{T'}[\widehat{\rho}'] \cap \widehat{NPR}_T[t](\widehat{\rho}) \mid compatible(\widehat{\rho}, \widehat{\rho}')\}$
5: **end for**
6: **end for**
7: return $NDIF_T[t]$

Complexity Analysis. The best known upper bound on the time complexity of our algorithm is quadratic in the number of threads, quadratic in the size of the domain, linear in the number of transitions in each thread, and double exponential in the number of locks. We stress that this is a *worst-case* bound, and we expect our algorithm to perform considerably better in practice. Programmers tend to follow certain disciplines when using locks, which decreases the double exponential factor in our algorithm. For example, allowing only constant-depth nesting of locks reduces the factor to single exponential. In practice, locksets, nesting depths, and consequently acquisition history sizes are very small (even if the number of locks in the program is not very small); and the complexity of our algorithm depends on the average size of acquisition histories, and not directly on the number of locks. Our experimental results in Section 5 confirm that our algorithm does indeed perform very well on real programs.

5 A Case Study

We implemented the intraprocedural version of our algorithm and evaluated its performance on a nontrivial concurrent program. Our experiments indicate that our algorithm scales well in practice; in particular, its performance appears to be only weakly dependent on the number of threads. This is remarkable, considering the program analysis community's historical difficulties with multithreaded code.

The algorithm is implemented in OCaml, and is applicable to C programs using the *pthreads* library for thread operations. We use the CIL program analysis infrastructure for parsing, CFG construction, and sequential data flow analysis. The algorithm is parameterized by a module that specifies the gen/kill sets for each instruction, so lifting sequential bitvector analyses to handle threads and locking is completely automatic. We implemented a reaching definitions analysis module and instantiated our concurrent bitvector analysis with it; this concurrent reaching definitions analysis was the subject of our evaluation.

We evaluated the performance of our algorithm on FUSE, a Unix kernel module and library that allows filesystems to be implemented in userspace programs. FUSE exposes parts of the kernel that are relevant to filesystems to userspace programs, essentially acting as a bridge between userspace and kernelspace. We analyzed the userspace portion.

Table 1. Experimental Results for FUSE

Test	$\lvert\mathcal{T}\rvert$	$\lvert N\rvert$	$\lvert\Sigma\rvert$	$\lvert\mathbb{D}\rvert$	$\lvert\mathcal{L}\rvert$	Time
5 avg	5	2568.1	3037.9	208.9	1.0	1.0
5 med	5	405.0	453.0	62.0	1.0	0.1
10 avg	10	4921.9	5820.1	401.6	1.3	1.8
10 med	10	988.5	1105.0	155.5	1.0	0.1
50 avg	50	24986.3	29546.0	2047.0	3.1	10.7
50 med	50	22628.5	26607.0	2022.5	3.0	4.6
200a	200	79985	94120	6861	6	36.4
200b	200	119905	142248	9515	4	116.5
full	425	218284	258219	17760	6	347.8

Since our implementation currently supports only intraprocedural analyses, we inlined all of the procedures defined within FUSE and ignored calls to library procedures that did not acquire or release locks. We did a type-based must-alias analysis to create a finite version of the set of locks and shared variables. Some procedures in the program had the (implicit) precondition that callers must hold a particular lock or set of locks at each call site; these 35 procedures could not be considered to be threads because they did not respect nested locking when considered independently. Each of the remaining 425 procedures was considered to be a distinct thread in our analysis. We divided these procedures into groups of 5 procedures, and analyzed each of those separately (that is, we analyzed the program consisting of procedures 1-5, 6-10, 11-15, etc). We repeated this process with groups of 10, 50, 100, 200, and also analyzed the entire program. We present mean and median statistics for the groups of 5, 10, 50, and 100 procedures. The experiments were conducted on a 3.16 GHz Linux machine with 4GB of memory.

Table 1 presents the results of our experiments. The $|\mathcal{T}|$, $|N|$, $|\Sigma|$, $|\mathbb{D}|$, $|\mathcal{L}|$, and Time columns indicate number of threads, number of CFA nodes, number of CFA transitions, number of data flow facts, number of locks, and running time (in seconds), respectively. As a result of the inlining step, there was a very large size gap between the smallest and the largest procedures that we analyzed, which we believe accounts for the discrepancy between the mean and median statistics.

Our thread summarization technique is a very effective optimization in practice. As an example, for a scenario with 123 threads, and a CFA size of approximately 200K (sum of the number of nodes and transitions), the analysis time is 50 seconds with summarization, while it is 930 seconds without summarization – *about 20 times slower*.

In Figure 3(a), we observe that the running time of our algorithm appears to grow quadratically in the number of threads in the program. However, the dispersion is quite high, which suggests that the running time has a weak relationship with the number of threads in the program. Indeed, the apparent quadratic relationship can be explained by the fact that the points that contain more threads also contain more total CFA transitions. Figure 3(b) shows the

running time of our algorithm as a function of total number of CFA transitions in the program, which is a much tighter fit.

Figure 3(c) shows the running time of our algorithm as a function of the product of the number of CFA transitions and the domain size of the program. This relationship is interesting because the time complexity of sequential bitvector analysis is $O(|\Sigma| \cdot |\mathbb{D}|)$. Our results indicate that there is a linear relationship between the running time of our algorithm and the product of the number of CFA transitions and domain size of the program, which suggests that our algorithm's running time is proportional to $|\Sigma| \cdot |\mathbb{D}|$ in practice.

Our empirical analysis is not completely rigorous. In particular, our data points are not independent and our treatment of memory locations is not conservative. However, we believe that the results obtained are promising and suggest that the algorithm can be used as the basis for further work on data flow analysis for concurrent programs.

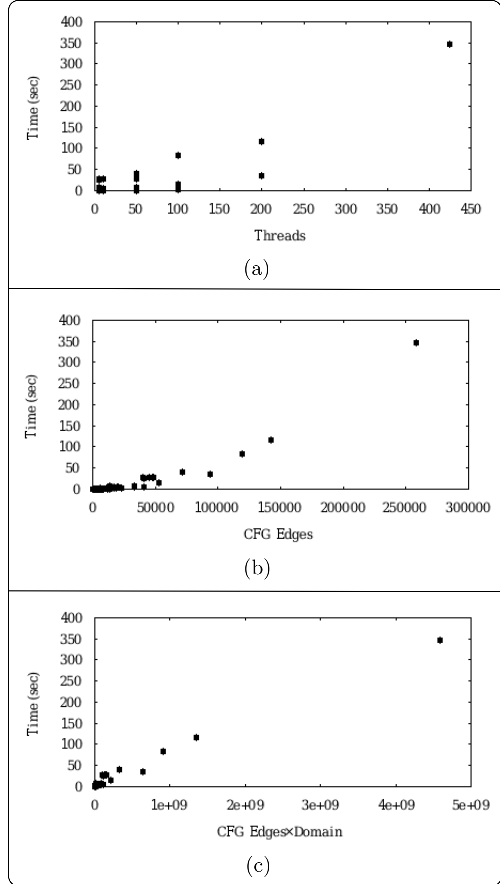


Fig. 3. Running time

6 Application and Future Work

We discussed a number of very important applications of bitvector analysis in Section 1. One of the most exciting applications of our *precise* bitvector framework (in our opinion) is our ongoing work on studying more suitable abstractions for concurrent programs. Intuitively, by computing the solution to reaching-definitions analysis for a concurrent program, we can collect information about how program threads interact. We are currently working on using this information to construct abstractions to be used for more powerful concurrent program analyses, such as computing state invariants for concurrent libraries. The precision offered by our concurrent bitvector analysis approach is quite important in this domain, because it affects both the precision of the invariants that can be computed, and the efficiency of their computation.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1986)
2. Chugh, R., Voung, J., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: PLDI, pp. 316–326 (2008)
3. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural dataflow analysis. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 14–30. Springer, Heidelberg (1999)
4. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: POPL, pp. 1–11 (2000)
5. Farzan, A., Kincaid, Z.: Compositional bitvector analysis for concurrent programs with nested locks.Technical report, University of Toronto (2010), http://www.cs.toronto.edu/~zkincaid/pub/cbva.pdf
6. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL, pp. 303–314 (2007)
7. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
8. Kidd, N., Lammich, P., Touili, T., Reps, T.: A decision procedure for detecting atomicity violations for communicating processes with locks. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 125–142. Springer, Heidelberg (2009)
9. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. TOPLAS 18(3), 268–299 (1996)
10. Knoop, J.: Parallel constant propagation. In: Pritchard, D., Reeve, J.S. (eds.) Euro-Par 1998. LNCS, vol. 1470, pp. 445–455. Springer, Heidelberg (1998)
11. Krinke, J.: Static slicing of threaded programs. SIGPLAN Not. 33(7), 35–42 (1998)
12. Lammich, P., Müller-Olm, M.: Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 205–220. Springer, Heidelberg (2008)
13. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. ACM Commun. 18(12), 717–721 (1975)
14. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: PPOPP, New York, NY, USA, pp. 129–138 (1993)
15. Muchnick, S.S.: Advanced Compiler Design and Imlementation. Morgan Kaufmann, San Francisco (1997)
16. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that happen in parallel. In: FSE, pp. 24–34 (1998)
17. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing mhp information for concurrent java programs. In: ESEC/FSE-7, pp. 338–354 (1999)
18. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
19. Nielson, F., Nielson, H.: Type and effect systems. In: Correct System Design, pp. 114–136 (1999)

20. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
21. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP (2001)
22. Seidl, H., Steffen, B.: Constraint-based inter-procedural analysis of parallel programs. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 351–365. Springer, Heidelberg (2000)

Computing Relaxed Abstract Semantics w.r.t. Quadratic Zones Precisely*

Thomas Martin Gawlitza[1,**] and Helmut Seidl[2]

[1] CNRS/VERIMAG
Thomas.Gawlitza@imag.fr
[2] TU München, Institut für Informatik, I2, 85748 München, Germany
seidl@in.tum.de

Abstract. In the present paper we compute numerical invariants of programs by abstract interpretation. For that we consider the abstract domain of *quadratic zones* recently introduced byAdjé et al. [2]. We use a relaxed abstract semantics which is at least as precise as the relaxed abstract semantics ofAdjé et al. [2]. For computing our relaxed abstract semantics, we present a practical strategy improvement algorithm for precisely computing least solutions of fixpoint equation systems, whose right-hand sides use order-concave operators and the maximum operator. These fixpoint equation systems strictly generalize the fixpoint equation systems considered by Gawlitza and Seidl [11].

1 Introduction

In the present paper we develop a practical strategy improvement algorithm for precisely computing least solutions of systems of in-equations of the form

$$\mathbf{x} \geq f(\mathbf{x}_1, \ldots, \mathbf{x}_k),$$

where f is an arbitrary *monotone and order-concave* operator on $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$ and $\mathbf{x}, \mathbf{x}_1, \ldots, \mathbf{x}_k$ are variables that take values in $\overline{\mathbb{R}}$. If the operators can be implemented through *parametrized* semidefinite programs, then we can implement the strategy improvement step using semidefinite programming. We finally show how to apply our techniques for computing a relaxed abstract semantics which is at least as precise as the relaxed abstract semantics of Adjé et al. [2]. We emphasize that, in contrast to Adjé et al. [2], we compute our relaxed abstract semantics *precisely*. Moreover, our algorithm always terminates after at most exponentially many strategy improvement steps.

Costan et al. [6] introduced a general *strategy iteration schema* (also called *policy iteration schema*) for computing fixpoints of monotone self-maps f, where $f = \min_{i \in I} f_i$ and, for every $i \in I$, the least fixpoint μf_i of f_i can be computed efficiently. The $f_i's$ are the policies. Starting with an arbitrary policy f_{i_0}, the policy is successively improved. The sequence of attained policies f_{i_k} results in a decreasing sequence $\mu f_{i_0} > \mu f_{i_1} > \ldots > \mu f_{i_k}$. Here, μg denotes the least fixpoint of a monotone self-map g. The algorithm stops whenever μf_{i_k} is a fixpoint of f — not necessarily the least

* This work was partially funded by the ANR project ASOPT.
** VERIMAG is a joint laboratory of CNRS, Université Joseph Fourier and Grenoble INP.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 271–286, 2010.
© Springer-Verlag Berlin Heidelberg 2010

one. Furthermore, this method does not necessarily reach a fixpoint after finitely many steps, whenever I is infinite. The quality of the obtained fixpoint may also depend on the initial policy f_{i_0}. Costan et al. [6] showed how to use their framework for performing interval analysis without widening. Gaubert et al. [10] extended this work to the following *relational* abstract domains: The *zone domain* [14], the *octagon domain* [15] and in particular the *template polyhedra domain* [18]. Gawlitza and Seidl [11] presented a practical strategy improvement algorithm for *precisely* computing least solutions of *systems of rational equations*. Their strategy improvement algorithm enables them to perform a precise template polyhedra analysis as well. It computes the least solution of the abstract semantic equations — not just some solution. Their algorithm for systems of rational equations can also be applied for analyzing *recursive stochastic games* [7–9, 20].

Recently, Adjé et al. [2] introduced a new *abstract domain* (called *zones*) for finding subtle numerical invariants of programs by abstract interpretation. This domain consists of linear and non-linear templates and thus generalizes the template polyhedra domain of Sankaranarayanan et al. [18]. Adjé et al. [2] considered a *relaxed abstract semantics* w.r.t. the new domain under the assumption that all templates are quadratic (*quadratic zones*). Their relaxed abstract semantics is based on Shor's semidefinite relaxation schema. The authors then solved the resulting fixpoint equations using their policy iteration schema, i.e., they coupled policy iteration and semidefinite programming. The policies in their approach are given by the Lagrange multipliers introduced by Shor's relaxation schema. This implies that there are potentially uncountably many policies. Accordingly, their algorithm does not terminate in all cases. Moreover, even if it terminates, the obtained solution is not necessarily the least solution. Still, the algorithm seems to be quite useful in many cases as it can be stopped at any time with a sound over-approximation of the least solution.

In the present paper we use the *semidefinite dual* of Shor's relaxation schema (as used by Adjé et al. [2]) for defining our relaxed abstract semantics. One consequence is that our relaxed abstract semantics is at least as precise as the relaxed abstract semantics used by Adjé et al. [2]. In addition it allows us to apply the strategy improvement approach of Gawlitza and Seidl [11, 12, 13] which iterates over *max-strategies*. Considering max-strategies instead of min-strategies (as done by Adjé et al. [2]) has the advantage that there exist only finitely (to be more precise exponentially) many max-strategies. We develop a strategy improvement algorithm which iterates over max-strategies and returns *precise* least solutions at the latest after considering all max-strategies. Each strategy improvement step can be realized by solving linearly many semidefinite programming problems. Our strategy improvement algorithm can be applied to *systems of order-concave equations* with finite least solutions. The restriction to finite least solutions, however, can be eliminated. The class of order-concave equations strictly generalizes the class of rational equations introduced by Gawlitza and Seidl [11]. Accordingly, the strategy improvement algorithm presented in the present paper strictly generalizes the strategy improvement algorithm of Gawlitza and Seidl [11]. In fact, for systems of rational LP-equations, it will perform the same strategy improvement steps. Moreover, the SDP's that have to be solved for each strategy improvement step degenerate to linear programming problems in this case.

2 Basics

Notations. The set of real numbers (resp. the set of rational numbers) is denoted by $\mathbb{R}$ (resp. $\mathbb{Q}$). The complete linear ordered set $\mathbb{R} \cup \{-\infty, \infty\}$ is denoted by $\overline{\mathbb{R}}$. Additionally, we set $\overline{\mathbb{Q}} := \mathbb{Q} \cup \{-\infty, \infty\}$. We call two vectors $x, y \in \overline{\mathbb{R}}^n$ *comparable* iff $x \leq y$ or $y \leq x$ holds. For $f : X \to \overline{\mathbb{R}}^m$ with $X \subseteq \overline{\mathbb{R}}^n$, we set $\mathrm{dom}(f) := \{x \in X \mid f(x) \in \mathbb{R}^m\}$ and $\mathrm{fdom}(f) := \mathrm{dom}(f) \cap \mathbb{R}^n$. We denote the i-th row (resp. j-th column) of a matrix A by $A_{i\cdot}$ (resp. $A_{\cdot j}$). Accordingly, $A_{i \cdot j}$ denotes the component in the i-th row and the j-th column. We also use this notation for vectors and functions $f : X \to Y^k$. $S\mathbb{R}^{n \times n}$ (resp. $S\mathbb{R}_+^{n \times n}$) denotes the set of symmetric matrices (resp. the set of positive semidefinite matrices). The square root of a positive semidefinite matrix X is denoted by $X^{\frac{1}{2}}$. $\preceq$ denotes the Löwner ordering of symmetric matrices, i.e., $A \preceq B$ iff $B - A \in S\mathbb{R}_+^{n \times n}$. $\mathrm{Tr}(A)$ denotes the trace of a square matrix $A \in \mathbb{R}^{n \times n}$, i.e., $\mathrm{Tr}(A) = \sum_{i=1}^n A_{i \cdot i}$. The inner product of two matrices A and B is denoted by $A \bullet B$, i.e., $A \bullet B = \mathrm{Tr}(A^\top B)$. For $\mathcal{A} = (A_1, \ldots, A_m)$ with $A_i \in \mathbb{R}^{n \times n}$ for all $i = 1, \ldots, m$, we denote the vector $(A_1 \bullet X, \ldots, A_m \bullet X)^\top$ by $\mathcal{A}(X)$. For $x \in \mathbb{R}^n$, the dyadic matrix $X(x)$ is defined by $X(x) := (1, x^\top)^\top (1, x^\top)$. The standard base vectors of $\mathbb{R}^n$ are denoted by $e_1, \ldots, e_n$.

(Order-)Convex/(Order-)Concave Functions. A set $X \subseteq \mathbb{R}^n$ is called *order-convex* iff $\lambda x + (1 - \lambda) y \in X$ holds for all comparable $x, y \in X$ and all $\lambda \in [0, 1]$. It is called *convex* iff this condition holds for all $x, y \in X$. A mapping $f : X \to \mathbb{R}^m$ with $X \subseteq \mathbb{R}^n$ order-convex is called *order-convex* (resp. *order-concave*) iff $f(\lambda x + (1 - \lambda) y) \leq$ (resp. $\geq$) $\lambda f(x) + (1 - \lambda) f(y)$ holds for all comparable $x, y \in M$ and all $\lambda \in [0, 1]$ (cf. Ortega and Rheinboldt [17]). A mapping $f : X \to \mathbb{R}^m$ with $X \subseteq \mathbb{R}^n$ convex is called *convex* (resp. *concave*) iff $f(\lambda x + (1 - \lambda) y) \leq$ (resp. $\geq$) $\lambda f(x) + (1 - \lambda) f(y)$ holds for all $x, y \in M$ and all $\lambda \in [0, 1]$ (cf. Ortega and Rheinboldt [17]). Every convex (resp. concave) function is order-convex (resp. order-concave). Note that f is (order-)concave iff $-f$ is (order-)convex. Note also that f is (order-)convex (resp. (order-)concave) iff $f_{i\cdot}$ is (order-)convex (resp. (order-)concave) for all $i = 1, \ldots, m$. We extend the notion of (order-)convexity/(order-)concavity to $\overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ as follows: A mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ is called *(order-)convex* (resp. *(order-)concave*) iff $\mathrm{fdom}(f)$ is (order-)convex and $f|_{\mathrm{fdom}(f)}$ is (order-)convex (resp. (order-)concave). Note that $\mathrm{fdom}(f)$ is order-convex for all monotone mappings $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$. In the following we are only concerned with mappings $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ which are monotone and order-concave.

Lemma 1. *Every linear function is concave and convex. The operator $\vee$ is convex, but not order-concave. The operator $\wedge$ is concave, but not order-convex.* □

Semidefinite Programming. We consider semidefinite programming problems (SDP problems for short) of the form

$$\sup\{C \bullet X \mid X \in S\mathbb{R}_+^{n \times n}, \mathcal{A}(X) = a, \mathcal{B}(X) \leq b\},$$

where $\mathcal{A} = (A_1, \ldots, A_m)$, $a \in \mathbb{R}^m$, $A_1, \ldots, A_m \in S\mathbb{R}^{n \times n}$, $\mathcal{B} = (B_1, \ldots, B_k)$, $B_1, \ldots, B_k \in S\mathbb{R}^{n \times n}$, $b \in \mathbb{R}^k$, and $C \in S\mathbb{R}^{n \times n}$. The set $\{X \in S\mathbb{R}_+^{n \times n} \mid \mathcal{A}(X) = a, \mathcal{B}(X) \leq b\}$ is called the *feasible space*. An element of the feasible space, is called *feasible solution*.

For $\mathcal{A} = (A_1, \ldots, A_m)$, $A_1, \ldots, A_m \in S\mathbb{R}^{n \times n}$, $a \in \mathbb{R}^m$, $\mathcal{B} = (B_1, \ldots, B_k)$, $B_1, \ldots, B_k \in S\mathbb{R}^{n \times n}$, and $C \in S\mathbb{R}^{n \times n}$ we define the operator $\mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C} : \overline{\mathbb{R}}^k \to \overline{\mathbb{R}}$ which solves a *parametrized* SDP problem by:

$$\mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}(b) := \sup\{C \bullet X \mid X \in S\mathbb{R}_+^{n \times n}, \mathcal{A}(X) = a, \mathcal{B}(X) \leq b\}, \quad b \in \overline{\mathbb{R}}^k$$

The SDP-operators generalize the LP-operators used by Gawlitza and Seidl [11, 13].

Lemma 2. $\mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}$ *is monotone and concave. If* $\mathrm{fdom}(\mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}) \neq \emptyset$*, then* $\mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}(b) < \infty$ *for all* $b \in \mathbb{R}^k$. □

Proof. Let $f := \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}$. The fact that f is monotone is obvious. Firstly, we show that $f(b) < \infty$ holds for all $b \in \mathbb{R}^k$, whenever $\mathrm{fdom}(f) \neq \emptyset$. For the sake of contradiction assume that there exist $b_1, b_2 \in \mathbb{R}^k$ such that $f(b_1) \in \mathbb{R}$ and $f(b_2) = \infty$ hold. Note that $M_i := \{X \in S\mathbb{R}_+^{n \times n} \mid \mathcal{A}(X) = a, \mathcal{B}(X) \leq b_i\}$ are convex for $i = 1, 2$. Thus there exists some $D \in S\mathbb{R}_+^{n \times n}$ such that $C \bullet D > 0$ and $M_2 + \{\lambda D \mid \lambda \in \mathbb{R}_{\geq 0}\} \subseteq M_2$ hold. Therefore $\mathcal{A}(D) = 0$ and $\mathcal{B}(D) \leq 0$.

Let $X_1 \in S\mathbb{R}_+^{n \times n}$ with $\mathcal{A}(X_1) = a$ and $\mathcal{B}(X_1) \leq b_1$. Then $\mathcal{A}(X_1 + \lambda D) = \mathcal{A}(X_1) + \lambda \mathcal{A}(D) = a$ and $\mathcal{B}(X_1 + \lambda D) = \mathcal{B}(X_1) + \lambda \mathcal{B}(D) \leq b_1$ hold for all $\lambda > 0$. Thus $f(b_1) = \infty$ — contradiction. Thus $f(b) < \infty$ holds for all $b \in \mathbb{R}^k$, whenever $\mathrm{fdom}(f) \neq \emptyset$.

In order to show that f is concave, we have to show that $\mathrm{fdom}(f)$ is convex and that $f|_{\mathrm{fdom}(f)}$ is concave. Assume that $\mathrm{fdom}(f) \neq \emptyset$. Thus $f(b) < \infty$ for all $b \in \mathbb{R}^k$. Let $b_1, b_2 \in \mathrm{fdom}(f)$, $\lambda \in [0, 1]$, and $b := \lambda b_1 + (1 - \lambda) b_2$. For all $b' \in \mathbb{R}^k$, let $M(b') := \{X \in S\mathbb{R}_+^{n \times n} \mid \mathcal{A}(X) = a, \mathcal{B}(X) \leq b'\}$. In order to show that

$$\lambda M(b_1) + (1 - \lambda) M(b_2) \subseteq M(b) \tag{1}$$

holds, let $X_i \in M(b_i)$, $i = 1, 2$, and $X = \lambda X_1 + (1 - \lambda) X_2$. Thus $X_i \in S\mathbb{R}_+^{n \times n}$, $\mathcal{A}(X_i) = a$, and $\mathcal{B}(X_i) \leq b_i$ for all $i = 1, 2$. Then $X \in S\mathbb{R}_+^{n \times n}$, $\mathcal{A}(X) = \lambda \mathcal{A}(X_1) + (1 - \lambda) \mathcal{A}(X_2) = a$, $\mathcal{B}(X) = \lambda \mathcal{B}(X_1) + (1 - \lambda) \mathcal{B}(X_1) \leq \lambda b_1 + (1 - \lambda) b_2 = b$. Therefore $X \in M(b)$. Using (1), we finally get:

$$\begin{aligned} f(b) &= \sup\{C \bullet X \mid X \in M(b)\} \\ &\geq \lambda \sup\{C \bullet X_1 \mid X_1 \in M(b_1)\} + (1 - \lambda) \sup\{C \bullet X_2 \mid X_2 \in M(b_2)\} \\ &= \lambda f(b_1) + (1 - \lambda) f(b_2) > -\infty \end{aligned}$$

Thus f is concave. □

The next example shows that the SDP-operator also includes the square root operator as a special case:

Example 1. We set $\sqrt{b} := \sup\{x \in \mathbb{R} \mid x^2 \leq b\}$ for all $b \in \overline{\mathbb{R}}$. Note that $\sqrt{b} = -\infty$ holds for all $b < 0$. Moreover, $\sqrt{\infty} = \infty$. Let

$$\mathcal{A} := \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \right), \quad a := 1, \quad \mathcal{B} := \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right), \quad C := \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}.$$

For $x, b \in \mathbb{R}_{\geq 0}$, $x^2 \leq b$ is equivalent to $\exists b'.x^2 \leq b' \leq b$. By the Schur complement theorem (c.f. section 3, example 5 of [19], for instance), this is equivalent to

$$\exists b'. \begin{pmatrix} 1 & x \\ x & b' \end{pmatrix} \succeq 0, \; b' \leq b.$$

This is equivalent to $\exists X \in S\mathbb{R}_+^{2\times 2}.x = X_{1\cdot 2} = X_{2\cdot 1}, \; \mathcal{A}(X) = a, \; \mathcal{B}(X) \leq b$. Thus, $\sqrt{b} = \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}(b)$ holds for all $b \in \overline{\mathbb{R}}$. $\square$

3 Solving Systems of Order-Concave Equations

Systems of Order-Concave Equations. Assume that a fixed set $\mathbf{X}$ of variables and a domain $\mathbb{D}$ is given. We consider equations of the form $\mathbf{x} = e$, where $\mathbf{x} \in \mathbf{X}$ is a variable and e is an expression over $\mathbb{D}$. A *system* $\mathcal{E}$ of equations is a finite set $\{\mathbf{x}_1 = e_1, \ldots, \mathbf{x}_n = e_n\}$ of equations, where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are pairwise distinct variables. We denote the set $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of variables occurring in $\mathcal{E}$ by $\mathbf{X}_{\mathcal{E}}$. We drop the subscript, whenever it is clear from the context.

The set of sub-expressions occurring in the right-hand sides of $\mathcal{E}$ is denoted by $\mathcal{S}(\mathcal{E})$. For a k-ary function f, $\mathcal{S}_f(\mathcal{E})$ denotes the set of all f-sub-expressions (sub-expressions of the form $f(e_1, \ldots, e_k)$) occurring in the right-hand sides of $\mathcal{E}$.

For a variable assignment $\rho : \mathbf{X} \to \mathbb{D}$, an expression e is mapped to a value $[\![e]\!]\rho$ by setting $[\![\mathbf{x}]\!]\rho := \rho(\mathbf{x})$ and $[\![f(e_1, \ldots, e_k)]\!]\rho := f([\![e_1]\!]\rho, \ldots, [\![e_k]\!]\rho)$, where $\mathbf{x} \in \mathbf{X}$, f is a k-ary operator ($k = 0$ is possible; then f is a constant), for instance $+$, and $e_1, \ldots, e_k$ are expressions. Let $\mathcal{E}$ be a system of equations. We define the unary operator $[\![\mathcal{E}]\!]$ on $\mathbf{X} \to \mathbb{D}$ by setting $([\![\mathcal{E}]\!]\rho)(\mathbf{x}) := [\![e]\!]\rho$ for all $\mathbf{x} = e \in \mathcal{E}$. A solution is a variable assignment ρ such that $\rho = [\![\mathcal{E}]\!]\rho$ holds. The set of solutions is denoted by $\mathbf{Sol}(\mathcal{E})$.

In the present paper, $\mathbb{D}$ will always be a complete lattice. Thus, assume in the following that $\mathbb{D}$ is a complete lattice. We denote the *least upper bound* and the *greatest lower bound* of a set X by $\bigvee X$ and $\bigwedge X$, respectively. The least element $\bigvee \emptyset$ (resp. the greatest element $\bigwedge \emptyset$) is denoted by $\bot$ (resp. $\top$). Accordingly, we define the binary operators $\vee$ and $\wedge$ by $x \vee y := \bigvee\{x, y\}$ and $x \wedge y := \bigwedge\{x, y\}$ for $x, y \in \mathbb{D}$, respectively. For $\square \in \{\vee, \wedge\}$, we will also consider $x_1 \;\square\; \cdots \;\square\; x_k$ as the application of a k-ary operator. This will cause no problems, since the binary operators $\vee$ and $\wedge$ are associative and commutative. An expression e (resp. an equation $\mathbf{x} = e$) is called *monotone* iff all operators occurring in e are monotone.

The set $\mathbf{X} \to \mathbb{D}$ of all *variable assignments* is a complete lattice. For $\rho, \rho' : \mathbf{X} \to \mathbb{D}$, we write $\rho \lhd \rho'$ (resp. $\rho \rhd \rho'$) iff $\rho(\mathbf{x}) < \rho'(\mathbf{x})$ (resp. $\rho(\mathbf{x}) > \rho'(\mathbf{x})$) holds for all $\mathbf{x} \in \mathbf{X}$. For $d \in \mathbb{D}$, $\underline{d}$ denotes the variable assignment $\{\mathbf{x} \mapsto d \mid \mathbf{x} \in \mathbf{X}\}$. A variable assignment ρ with $\underline{\bot} \lhd \rho \lhd \underline{\top}$ is called *finite*. A pre-solution (resp. post-solution) is a variable assignment ρ such that $\rho \leq [\![\mathcal{E}]\!]\rho$ (resp. $\rho \geq [\![\mathcal{E}]\!]\rho$) holds. The set of pre-solutions (resp. the set of post-solutions) is denoted by $\mathbf{PreSol}(\mathcal{E})$ (resp. $\mathbf{PostSol}(\mathcal{E})$). The least fixpoint (resp. the greatest fixpoint) of an operator $f : \mathbb{D} \to \mathbb{D}$ is denoted by μf (resp. νf), provided that it exists. Thus, the least solution (resp. the greatest solution) of a system $\mathcal{E}$ of equations is denoted by $\mu[\![\mathcal{E}]\!]$ (resp. $\nu[\![\mathcal{E}]\!]$), provided that it exists. For a pre-solution ρ (resp. for a post-solution ρ), $\mu_{\geq\rho}[\![\mathcal{E}]\!]$ (resp. $\nu_{\leq\rho}[\![\mathcal{E}]\!]$) denotes the

least solution that is greater than or equal to ρ (resp. the greatest solution that is less than or equal to ρ). In our setting, Knaster-Tarski's fixpoint theorem can be stated as follows: Every system $\mathcal{E}$ of monotone equations over a complete lattice has a least solution $\mu[\![\mathcal{E}]\!]$ and a greatest solution $\nu[\![\mathcal{E}]\!]$. Furthermore, $\mu[\![\mathcal{E}]\!] = \bigwedge \mathbf{PostSol}(\mathcal{E})$ and $\nu[\![\mathcal{E}]\!] = \bigvee \mathbf{PreSol}(\mathcal{E})$.

Definition 1 (Order-Concave Equations). *An expression* e *(resp. equation* $\mathbf{x} = e$*) over* $\overline{\mathbb{R}}$ *is called* basic order-concave expression *(resp.* basic order-concave equation*) iff* $[\![e]\!]$ *is monotone and order-concave and* $[\![e]\!]\rho < \infty$ *holds for all* $\rho : \mathbf{X} \to \mathbb{R}$*, whenever* $\mathrm{fdom}([\![e]\!]) \neq \emptyset$*. An expression* e *(resp. equation* $\mathbf{x} = e$*) over* $\overline{\mathbb{R}}$ *is called* order-concave *iff* $e = e_1 \vee \cdots \vee e_k$*, where* $e_1, \ldots, e_k$ *are basic order-concave expressions.* □

Note that by lemma 1 the class of systems of order-concave equations strictly subsumes the class of *systems of rational equations* and even the class of *systems of rational LP-equations* as defined by Gawlitza and Seidl [11, 13].

Example 2. By lemma 2 and example 1, $\sqrt{\cdot}$ is monotone and concave, and $\sqrt{x} < \infty$ holds for all $x \in \mathbb{R}$. The least solution of the system $\mathcal{E} = \{\mathbf{x} = \frac{1}{2} \vee \sqrt{\mathbf{x}}\}$ of order-concave equations is $\mu[\![\mathcal{E}]\!] = 1$. □

Strategies. A $\vee$-*strategy* σ for $\mathcal{E}$ is a function that maps every expression $e_1 \vee \cdots \vee e_k$ occurring in $\mathcal{E}$ to one of the immediate sub-expressions e_j, $j \in \{1, \ldots, k\}$. We denote the set of all $\vee$-strategies for $\mathcal{E}$ by $\Sigma_{\mathcal{E}}$. We drop the subscript, whenever it is clear from the context. For $\sigma \in \Sigma$ the expression $e\sigma$ is inductively defined by

$$(e_1 \vee \cdots \vee e_k)\sigma := (\sigma(e_1 \vee \cdots \vee e_k))\sigma, \qquad (f(e_1, \ldots, e_k))\sigma := f(e_1\sigma, \ldots, e_k\sigma),$$

where $f \neq \vee$ is some operator. Finally, we set $\mathcal{E}(\sigma) := \{\mathbf{x} = e\sigma \mid \mathbf{x} = e \in \mathcal{E}\}$.

The Strategy Improvement Algorithm. We briefly explain the strategy improvement algorithm (cf. Gawlitza and Seidl [13]). In the present paper we are going to compute least solutions of systems of order-concave equations through our strategy improvement algorithm. Systems of order-concave equations are in particular systems of monotone equations over the *complete linear ordered set* $\overline{\mathbb{R}}$. For the sake of generality, we subsequently consider an arbitrary complete linear ordered set.

Our strategy improvement algorithm iterates over $\vee$-strategies. It maintains a current $\vee$-strategy and a current *approximate* to the least solution. A so called *strategy improvement operator* is used for determining a next, improved $\vee$-strategy. Whether or not a $\vee$-strategy represents an *improvement* of the current $\vee$-strategy may depend on the current approximate:

Definition 2 (Improvements). *Let* $\mathcal{E}$ *be a system of monotone equations over a complete linear ordered set. Let* $\sigma, \sigma' \in \Sigma$ *be* $\vee$*-strategies for* $\mathcal{E}$ *and* ρ *be a pre-solution of* $\mathcal{E}(\sigma)$*. The* $\vee$*-strategy* σ' *is called* improvement of σ w.r.t. ρ *iff the following conditions are fulfilled: (1) If* $\rho \notin \mathbf{Sol}(\mathcal{E})$*, then* $[\![\mathcal{E}(\sigma')]\!]\rho > \rho$*. (2) For all* $\vee$*-expressions* $e \in \mathcal{S}_\vee(\mathcal{E})$ *the following holds: If* $\sigma'(e) \neq \sigma(e)$*, then* $[\![e\sigma']\!]\rho > [\![e\sigma]\!]\rho$*. A function* $P_\vee$ *which assigns an improvement of* σ *w.r.t.* ρ *to every pair* (σ, ρ)*, where* σ *is a* $\vee$*-strategy and* ρ *is a pre-solution of* $\mathcal{E}(\sigma)$*, is called* $\vee$-strategy improvement operator. □

In many cases, there exist several, different improvements of a $\vee$-strategy σ w.r.t. a pre-solution ρ of $\mathcal{E}(\sigma)$. Accordingly, there exist several, different strategy improvement operators. Under the assumption that the operator $\vee$ is only used in its binary version, one is known as *all profitable switches* (see e.g. Björklund et al. [3, 4]). Carried over to the case considered here, this means, that the $\vee$-strategy σ will be modified at any $\vee$-expression $e_1 \vee e_2$ with $[\![e_1 \vee e_2]\!]\rho > [\![\sigma(e_1 \vee e_2)]\!]\rho$. According to definition 2 the selection at the other $\vee$-expressions must be preserved. The computation of this improvement is realized through the strategy improvement operator $P_\vee^{\mathsf{eager}}$:

Definition 3 (The Strategy Improvement Operator $P_\vee^{\mathsf{eager}}$). *Let $\mathcal{E}$ be a system of monotone equations over a complete linear ordered set, $\sigma \in \Sigma$ and $\rho \in \mathbf{PreSol}(\mathcal{E}(\sigma))$. The $\vee$-strategy $P_\vee^{\mathsf{eager}}(\sigma, \rho)$ is defined by*

$$P_\vee^{\mathsf{eager}}(\sigma,\rho)(e_1 \vee e_2) = \begin{cases} e_1 & \text{if } [\![e_1]\!]\rho > [\![e_2]\!]\rho \\ e_2 & \text{if } [\![e_1]\!]\rho < [\![e_2]\!]\rho \\ \sigma(e_1 \vee e_2) & \text{if } [\![e_1]\!]\rho = [\![e_2]\!]\rho \end{cases}, \quad e_1 \vee e_2 \in \mathcal{S}_\vee(\mathcal{E}). \qquad \square$$

Lemma 3 (Properties of $P_\vee^{\mathsf{eager}}$). *Let $\mathcal{E}$ be a system of monotone equations over a complete linear ordered set. The following statements hold:*

1. $[\![\mathcal{E}(P_\vee^{\mathsf{eager}}(\sigma,\rho))]\!]\rho = [\![\mathcal{E}]\!]\rho$ *holds for all* $\sigma \in \Sigma$ *and all* $\rho \in \mathbf{PreSol}(\mathcal{E}(\sigma))$.
2. $P_\vee^{\mathsf{eager}}$ *is a $\vee$-strategy improvement operator.* $\square$

If $\vee$ is not only used in its binary version, we can think of $\sigma' = P_\vee^{\mathsf{eager}}(\sigma,\rho)$ as some arbitrary improvement of σ w.r.t. ρ such that $[\![\mathcal{E}(\sigma')]\!]\rho = [\![\mathcal{E}]\!]\rho$ holds. Note that then $\sigma' = P_\vee^{\mathsf{eager}}(\sigma,\rho)$ is not necessarily uniquely determined. However, this is not important for the following results which refer to the strategy improvement operator $P_\vee^{\mathsf{eager}}$.

Now we can formulate the strategy improvement algorithm for computing least solutions of systems of monotone equations over complete linear ordered sets. This algorithm is parameterized with a $\vee$-strategy improvement operator $P_\vee$. The input is a system $\mathcal{E}$ of monotone equations over a complete linear ordered set, a $\vee$-strategy σ_{init} for $\mathcal{E}$, and a pre-solution ρ_{init} of $\mathcal{E}(\sigma_{\text{init}})$. In order to compute the *least* and not just some solution, we additionally claim that $\rho_{\text{init}} \leq \mu[\![\mathcal{E}]\!]$ holds:

Algorithm 1. The Strategy Improvement Algorithm

Parameter : A $\vee$-strategy improvement operator $P_\vee$

Input :
- A system $\mathcal{E}$ of monotone equations over a complete linear ordered set
- A $\vee$-strategy σ_{init} for $\mathcal{E}$
- A pre-solution ρ_{init} of $\mathcal{E}(\sigma_{\text{init}})$ with $\rho_{\text{init}} \leq \mu[\![\mathcal{E}]\!]$

Output : The least solution $\mu[\![\mathcal{E}]\!]$ of $\mathcal{E}$

$\sigma \leftarrow \sigma_{\text{init}};\ \rho \leftarrow \rho_{\text{init}};$
while $(\rho \notin \mathbf{Sol}(\mathcal{E}))$ $\{\sigma \leftarrow P_\vee(\sigma,\rho);\ \rho \leftarrow \mu_{\geq\rho}[\![\mathcal{E}(\sigma)]\!];\ \}$
return ρ;

Lemma 4. *Let $\mathcal{E}$ be a system of monotone equations over a complete linear ordered set. For $i \in \mathbb{N}$, let ρ_i be the value of the program variable ρ and σ_i be the value of the program variable σ in the strategy improvement algorithm (algorithm 1) after the i-th evaluation of the loop-body. The following statements hold for all $i \in \mathbb{N}$:*

1. $\rho_i \leq \mu[\![\mathcal{E}]\!]$. *2.* $\rho_i \in \mathbf{PreSol}(\mathcal{E}(\sigma_{i+1}))$.
3. If $\rho_i < \mu[\![\mathcal{E}]\!]$*, then* $\rho_{i+1} > \rho_i$. *4. If* $\rho_i = \mu[\![\mathcal{E}]\!]$*, then* $\rho_{i+1} = \rho_i$. □

An immediate consequence of lemma 4 is the following: Whenever the strategy improvement algorithm terminates, it computes the least solution $\mu[\![\mathcal{E}]\!]$ of $\mathcal{E}$.

In the following we are interested in solving systems of order-concave equations with *finite* least solutions. We show that in this case our strategy improvement algorithm terminates and returns the least solution at the latest after considering all strategies. Moreover, we give an important characterization for $\mu_{\geq\rho}[\![\mathcal{E}(\sigma)]\!]$.

Feasibility. As in [11–13] we need some notation of *feasibility*. However, here we have to find a purely analytic criterion. From our notion of *feasibility* we expect that the following two claims hold: (1) If ρ is a *feasible* pre-solution of the system $\mathcal{E}$ of *basic* order-concave equations, then $\mu_{\geq\rho}[\![\mathcal{E}]\!]$ is the greatest finite pre-solution of $\mathcal{E}$. (2) Feasibility will be preserved by strategy improvement steps, i.e., if ρ is a feasible solution of $\mathcal{E}(\sigma)$ and σ' is an improvement of σ w.r.t. ρ, then ρ is also a feasible pre-solution of $\mathcal{E}(\sigma')$.

A simple notion of feasibility is the following (we restrict our considerations to finite pre-solutions): A finite solution ρ of $\mathcal{E}$ is called *feasible* iff there exists some $\rho' \lhd \rho$ such that $\rho' \lhd [\![\mathcal{E}]\!]\rho'$ holds. A finite pre-solution ρ of $\mathcal{E}$ is called *feasible* iff $\mu_{\geq\rho}[\![\mathcal{E}]\!]$ is feasible. This notion of feasibility ensures claim 1. However, claim 2 does not hold. But it almost holds in the sense that it only fails to hold for degenerated cases. Moreover, also this notion of feasibility guarantees at least that we compute an over-approximation of the least solution, when we apply our method.

Before we fix the problem of this notion, we consider an example of a degenerated case, where feasibility as defined above is not preserved by a strategy improvement step.

Example 3. Consider the system $\mathcal{E} = \{\mathbf{x}_1 = \mathbf{x}_2 + 1 \wedge 0, \mathbf{x}_2 = -1 \vee \sqrt{\mathbf{x}_1}\}$ of order-concave equations. Let $\sigma_1 = \{-1 \vee \sqrt{\mathbf{x}_1} \mapsto -1\}$ and $\sigma_2 = \{-1 \vee \sqrt{\mathbf{x}_1} \mapsto \sqrt{\mathbf{x}_1}\}$ be the two $\vee$-strategies for $\mathcal{E}$. Let $\rho = \{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto -1\}$. Obviously, ρ is a feasible solution of $\mathcal{E}(\sigma_1) = \{\mathbf{x}_1 = \mathbf{x}_2 + 1 \wedge 0, \mathbf{x}_2 = -1\}$. The $\vee$-strategy σ_2 is an improvement of the $\vee$-strategy σ_1 w.r.t. ρ. However, we can show that $\rho = \{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto -1\}$ is not a feasible pre-solution of $\mathcal{E}(\sigma_2) = \{\mathbf{x}_1 = \mathbf{x}_2 + 1 \wedge 0, \mathbf{x}_2 = \sqrt{\mathbf{x}_1}\}$. We have $\rho^* := \mu_{\geq\rho}[\![\mathcal{E}(\sigma_2)]\!] = \{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto 0\}$. For any $\rho' \lhd \rho^*$, we have $[\![\mathcal{E}(\sigma_2)]\!]\rho'(\mathbf{x}_2) = -\infty$. Thus ρ^* is not a feasible solution and therefore ρ is not a feasible pre-solution.

The problem comes from the fact that we switch the strategy with $\mathbf{x}_1 = 0$, and there exists no right derivative of $\sqrt{\cdot}$ at 0, and moreover $\rho(\mathbf{x}_1) = \rho^*(\mathbf{x}_1)$. These cases are degenerated cases. □

We now extend the notion of feasibility in order to deal with degenerated cases:

Definition 4 (Feasibility). *Let $\mathcal{E}$ be a system of basic order-concave equations. A finite solution ρ of $\mathcal{E}$ is called* ($\mathcal{E}$-)feasible *iff there exists $\mathbf{X}_1, \mathbf{X}_2 \subseteq \mathbf{X}$ and some $k \in \mathbb{N}$ such that the following statements hold:*

1. $\mathbf{X}_1 \cup \mathbf{X}_2 = \mathbf{X}$*, and* $\mathbf{X}_1 \cap \mathbf{X}_2 = \emptyset$.
2. *There exists some $\rho' \lhd \rho|_{\mathbf{X}_1}$ such that $\rho' \mathbin{\dot\cup} \rho|_{\mathbf{X}_2}$ is a pre-solution of $\mathcal{E}$, and $\rho = [\![\mathcal{E}]\!]^k(\rho' \mathbin{\dot\cup} \rho|_{\mathbf{X}_2})$.*

3. *There exists some $\rho' \lhd \rho|_{\mathbf{X}_2}$ such that $\rho' \lhd ([\![\mathcal{E}]\!]^k(\rho|_{\mathbf{X}_1} \dot{\cup} \rho'))|_{\mathbf{X}_2}$.*

*A finite pre-solution ρ of $\mathcal{E}$ is called ($\mathcal{E}$-)*feasible *iff $\mu_{\geq\rho}[\![\mathcal{E}]\!]$ is a feasible finite solution of $\mathcal{E}$. A pre-solution $\rho \lhd \underline{\infty}$ is called feasible iff $e = -\infty$ for all $\mathbf{x} = e \in \mathcal{E}$ with $[\![e]\!]\rho = -\infty$, and $\rho|_{\mathbf{X}'}$ is a feasible finite pre-solution of $\{\mathbf{x} = e \in \mathcal{E} \mid \mathbf{x} \in \mathbf{X}'\}$, where $\mathbf{X}' := \{\mathbf{x} \mid \mathbf{x} = e \in \mathcal{E}, [\![e]\!]\rho > -\infty\}$.*

A system $\mathcal{E}$ of basic order-concave equations is called feasible *iff there exists a feasible solution ρ of $\mathcal{E}$.* □

Example 4. We consider the system $\mathcal{E} = \{\mathbf{x} = \sqrt{\mathbf{x}}\}$ of basic order-concave equations. For all $x \in \overline{\mathbb{R}}$, let $\underline{x} := \{\mathbf{x} \mapsto x\}$. We show that $\underline{0}$ is not a feasible solution of $\mathcal{E}$. Let $x < 0$. Thus $\underline{x} < \underline{0}$. First we consider the case that $\mathbf{X}_1 = \{\mathbf{x}\}$ and $\mathbf{X}_2 = \emptyset$ hold. Then $\underline{0} \neq \underline{-\infty} = [\![\mathcal{E}]\!]^k\underline{x}$ for all $k \geq 1$. We now consider the case that $\mathbf{X}_1 = \emptyset$ and $\mathbf{X}_2 = \{\mathbf{x}\}$ hold. Then $\underline{0} \not\lhd \underline{-\infty} = [\![\mathcal{E}]\!]^k\underline{x}$ for all $k \geq 1$. Thus the solution $\underline{0}$ is not feasible. $\underline{1}$ is a feasible solution, since $\underline{\frac{1}{2}} \lhd [\![\sqrt{\mathbf{x}}]\!]\underline{\frac{1}{2}}$ holds. Thus, $\underline{x}$ is a feasible pre-solution for all $x \in (0,1]$. Note that $\underline{1}$ is the only feasible finite solution of $\mathcal{E}$ and additionally the greatest finite pre-solution of $\mathcal{E}$. □

Example 5. We continue example 3. We want to verify that our new notion of feasibility (definition 4) helps for this example, i.e., we again consider the system $\mathcal{E}(\sigma_2) = \{\mathbf{x}_1 = \mathbf{x}_2 + 1 \wedge 0, \mathbf{x}_2 = \sqrt{\mathbf{x}_1}\}$ of basic order-concave equations. We show that $\rho := \{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto 0\}$ now is a feasible solution. Let $\mathbf{X}_1 = \{\mathbf{x}_2\}$, $\mathbf{X}_2 = \{\mathbf{x}_1\}$. Then we have $\{\mathbf{x}_2 \mapsto -1\} \lhd \rho|_{\mathbf{X}_1}$, and $\rho = [\![\mathcal{E}(\sigma_2)]\!]\{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto -1\} = [\![\mathcal{E}(\sigma_2)]\!](\{\mathbf{x}_2 \mapsto -1\} \dot{\cup} \rho|_{\mathbf{X}_2})$. We also have $\{\mathbf{x}_1 \mapsto -1\} \lhd \rho|_{\mathbf{X}_2}$ and $\{\mathbf{x}_1 \mapsto -1\} \lhd \{\mathbf{x}_1 \mapsto 0\} = ([\![\mathcal{E}(\sigma_2)]\!](\rho|_{\mathbf{X}_1} \dot{\cup} \{\mathbf{x}_1 \mapsto -1\}))|_{\mathbf{X}_2}$. Thus ρ is a feasible solution. Thus $\{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto x\}$ is a feasible pre-solution for all $x \in [-1, 0]$. The pre-solution $\{\mathbf{x}_1 \mapsto -\infty, \mathbf{x}_2 \mapsto -\infty\}$ is not feasible, since the right-hand sides evaluate to $-\infty$ under $\{\mathbf{x}_1 \mapsto -\infty, \mathbf{x}_2 \mapsto -\infty\}$. □

The following lemma is an immediate consequence of the definition:

Lemma 5. *Let $\mathcal{E}$ be a system of basic order-concave equations and ρ be a feasible pre-solution of $\mathcal{E}$. Every pre-solution ρ' of $\mathcal{E}$ with $\rho \leq \rho' \leq \mu_{\geq\rho}[\![\mathcal{E}]\!]$ is feasible.* □

Feasibility is preserved by performing strategy improvement steps:

Lemma 6. *Let $\mathcal{E}$ be a system of order-concave equations, σ be a $\vee$-strategy for $\mathcal{E}$, ρ be a feasible solution of $\mathcal{E}(\sigma)$, and σ' be an improvement of σ w.r.t. ρ. Then ρ is a feasible pre-solution of $\mathcal{E}(\sigma')$.* □

Example 6. We again continue example 3. Obviously, $\rho = \{\mathbf{x}_1 \mapsto 0, \mathbf{x}_2 \mapsto -1\}$ is a feasible solution of $\mathcal{E}(\sigma_1) = \{\mathbf{x}_1 = \mathbf{x}_2 + 1 \wedge 0, \mathbf{x}_2 = -1\}$. The $\vee$-strategy σ_2 is an improvement of the $\vee$-strategy σ_1 w.r.t. ρ. By lemma 6, ρ is also a feasible pre-solution of $\mathcal{E}(\sigma_2) = \{\mathbf{x}_1 = \mathbf{x}_2 + 1 \wedge 0, \mathbf{x}_2 = \sqrt{\mathbf{x}_1}\}$. The fact that ρ is a feasible pre-solution of $\mathcal{E}(\sigma_2)$ is also shown in example 5. □

The above two lemmas ensure that our strategy improvement algorithm stays in the feasible area, whenever it is started in the feasible area.

In order to start in the feasible area, we simply start the strategy improvement algorithm with the system $\mathcal{E} \vee -\infty := \{\mathbf{x} = e \vee -\infty \mid \mathbf{x} = e \in \mathcal{E}\}$, the $\vee$-strategy $\sigma_{\text{init}} = \{e \vee -\infty \mapsto -\infty \mid \mathbf{x} = e \in \mathcal{E}\}$ and the feasible pre-solution $\underline{-\infty}$ of $(\mathcal{E} \vee -\infty)(\sigma_{\text{init}})$.

It remains to develop a method for computing $\mu_{\geq\rho}[\![\mathcal{E}]\!]$ under the assumption that ρ is a feasible pre-solution of the system $\mathcal{E}$ of basic order-concave equations. The following lemma in particular states that for this purpose we in fact have to compute the greatest finite pre-solution of $\mathcal{E}$:

Lemma 7. *Let $\mathcal{E}$ be a feasible system of basic order-concave equations. Assume that $e \neq -\infty$ holds for all $\mathbf{x} = e \in \mathcal{E}$. There exists a greatest finite pre-solution ρ^* of $\mathcal{E}$ and ρ^* is the only feasible solution of $\mathcal{E}$. If ρ is a feasible pre-solution of $\mathcal{E}$, then $\rho^* = \mu_{\geq\rho}[\![\mathcal{E}]\!]$.* □

Termination. Lemma 7 implies that our strategy improvement algorithm has to consider each $\vee$-strategy at most once. Thus, we have shown the following theorem:

Theorem 1. *Let $\mathcal{E}$ be a system of order-concave equations with $\mu[\![\mathcal{E}]\!] \lhd \underline{\infty}$. Assume that we can compute the greatest finite pre-solution ρ_σ of each $\mathcal{E}(\sigma)$, provided that $\mathcal{E}(\sigma)$ is feasible. Our strategy improvement algorithm computes $\mu[\![\mathcal{E}]\!]$ and performs at most $|\Sigma| + |\mathbf{X}|$ strategy improvement steps.* □

Example 7. We consider the system $\mathcal{E} = \left\{\mathbf{x} = -\infty \vee \frac{1}{2} \vee \sqrt{\mathbf{x}} \vee \frac{7}{8} + \sqrt{\mathbf{x} - \frac{47}{64}}\right\}$ of order-concave equations. We start with a strategy σ_0 such that $\mathcal{E}(\sigma_0) = \{\mathbf{x} = -\infty\}$ holds and with the feasible solution $\rho_0 := \{\mathbf{x} \mapsto -\infty\}$ of $\mathcal{E}(\sigma_0)$. Since $\rho_0 \notin \mathbf{Sol}(\mathcal{E})$, we improve σ_0 to a strategy σ_1 w.r.t. ρ_0. Necessarily, we get $\mathcal{E}(\sigma_1) = \{\mathbf{x} = \frac{1}{2}\}$. Furthermore, $\rho_1 := \mu_{\geq\rho_0}[\![\mathcal{E}(\sigma_1)]\!] = \{\mathbf{x} \mapsto \frac{1}{2}\}$. Since $\sqrt{\frac{1}{2}} > \frac{1}{2}$ and $\frac{7}{8} + \sqrt{\frac{1}{2} - \frac{47}{64}} < \frac{1}{2}$ hold, we necessarily improve the strategy σ_1 w.r.t. ρ_1 to a strategy σ_2 with $\mathcal{E}(\sigma_2) = \{\mathbf{x} = \sqrt{\mathbf{x}}\}$. We get $\rho_2 := \mu_{\geq\rho_1}[\![\mathcal{E}(\sigma_2)]\!] = \{\mathbf{x} \mapsto 1\}$, since $\mathbf{Sol}(\mathcal{E}(\sigma_2)) = \{\{\mathbf{x} \mapsto -\infty\}, \{\mathbf{x} \mapsto 0\}, \{\mathbf{x} \mapsto 1\}, \{\mathbf{x} \mapsto \infty\}\}$. Since $\frac{7}{8} + \sqrt{1 - \frac{47}{64}} > \frac{7}{8} + \sqrt{1 - \frac{60}{64}} = \frac{9}{8} > 1$, we get $\mathcal{E}(\sigma_3) = \{\mathbf{x} = \frac{7}{8} + \sqrt{\mathbf{x} - \frac{47}{64}}\}$. Finally we get $\rho_3 := \mu_{\geq\rho_2}[\![\mathcal{E}(\sigma_3)]\!] = \{\mathbf{x} \mapsto 2\}$. The algorithm terminates, because ρ_3 solves $\mathcal{E}$. Therefore $\rho_3 = \mu[\![\mathcal{E}]\!]$. □

4 Systems of SDP-Equations

We call a system of order-concave equations that, besides the $\vee$-operator, only uses the SDP-operators a *system of SDP-equations*. Systems of SDP-equations generalize systems of rational LP-equations as introduced by Gawlitza and Seidl [11, 13]. In this section we use our strategy improvement algorithm for solving Systems of SDP-equations. Because of lemma 7, for that we have to develop a method for computing greatest finite pre-solutions of systems of basic SDP-equations.

Let $\mathcal{E}$ be a system of *basic* SDP-equations, i.e., each equation of $\mathcal{E}$ is of the form $\mathbf{x} = \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}(\mathbf{x}_1, \ldots, \mathbf{x}_k)$. We replace every equation $\mathbf{x} = \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}(\mathbf{x}_1, \ldots, \mathbf{x}_k)$

of $\mathcal{E}$ with the constraints

$$\mathbf{x} \leq C \bullet \mathbf{Y}, \quad \mathcal{A}(\mathbf{Y}) = a, \quad \mathcal{B}(\mathbf{Y}) \leq (\mathbf{x}_1, \ldots, \mathbf{x}_k)^\top, \quad \mathbf{Y} \succeq 0,$$

where $\mathbf{Y}$ is a symmetric matrix of fresh variables. We denote the resulting system of constraints by $\mathcal{C}_{\mathbf{SDP}}(\mathcal{E})$. Then, we have:

Lemma 8. *Let $\mathcal{E}$ be a system of basic SDP-equations with a greatest finite pre-solution ρ^*. Then $\rho^*(\mathbf{x}) = \sup \{\rho(\mathbf{x}) \mid \rho : \mathbf{X}_{\mathcal{C}_{\mathbf{SDP}}(\mathcal{E})} \to \mathbb{R},\ \rho \in \mathbf{Sol}(\mathcal{C}_{\mathbf{SDP}}(\mathcal{E}))\}$ holds for all $\mathbf{x} \in \mathbf{X}_{\mathcal{E}}$.* □

Because of Lemma 8, the greatest finite pre-solution ρ^*, provided that it exists, can be computed by solving $|\mathbf{X}|$ SDP problems, each of which can be obtained from $\mathcal{E}$ in linear time. In practice, this can be improved by solving the SDP problem

$$\sup \left\{\textstyle\sum_{\mathbf{x} \in \mathbf{X}} \rho(\mathbf{x}) \mid \rho : \mathbf{X}_{\mathcal{C}_{\mathbf{SDP}}(\mathcal{E})} \to \mathbb{R},\ \rho \in \mathbf{Sol}(\mathcal{C}_{\mathbf{SDP}}(\mathcal{E}))\right\}.$$

If this problem has an optimal solution, then it determines ρ^*. Moreover, if it has no optimal solution, then any ϵ-optimal solution of the above SDP problem determines a pre-solution of $\mathcal{E}$ that is ϵ-close to ρ^*. Let, for every $\epsilon > 0$, ρ_ϵ denote a pre-solution of $\mathcal{E}$ that is determined by some ϵ-optimal solution of the above SDP problem. Then $\lim_{\epsilon \to 0} \rho_\epsilon = \rho^*$. Since in many practical cases, we can anyway only approximate optimal values of SDP problems, in a practical implementation it thus suffices to solve the above SDP problem instead of the $|\mathbf{X}|$ SDP-problems mentioned in lemma 8.

Together with the result from section 3 we get the following result:

Theorem 2. *Let $\mathcal{E}$ be a system of SDP-equations with $\mu[\![\mathcal{E}]\!] \lhd \underline{\infty}$. Our strategy improvement algorithm returns $\mu[\![\mathcal{E}]\!]$ after performing at most $|\Sigma| + |\mathbf{X}|$ strategy improvement steps. Each strategy improvement step can be performed by solving linearly many SDP problems, each of which can be constructed in linear time.* □

Our techniques can be extended in order to get rid of the assumption that $\mu[\![\mathcal{E}]\!] \lhd \underline{\infty}$ holds. Basically, if the SDP problem that is solved for computing $\rho^*(\mathbf{x})$ in lemma 8 is unbounded, then we know that $\mu[\![\mathcal{E}]\!](\mathbf{x}) = \infty$ holds. However, for simplicity, we do not discuss these aspects in detail in the present paper.

5 Quadratic Zones and Relaxed Abstract Semantics

We consider statements of the form $x := Ax + b$ and $x^\top Ax + 2b^\top x \leq c$, where $A \in \mathbb{R}^{n \times n}$ (resp. $A \in S\mathbb{R}^{n \times n}$), $b \in \mathbb{R}^n$, $c \in \mathbb{R}$, and $x \in \mathbb{R}^n$ denotes the vector of program variables. Statements of the form $x := Ax + b$ are called *(affine) assignments*. Statements of the form $x^\top Ax + 2b^\top x \leq c$ are called *(quadratic) guards*. The set of statements is denoted by $\mathbf{Stmt}$. The *collecting semantics* $[\![s]\!] : 2^{\mathbb{R}^n} \to 2^{\mathbb{R}^n}$ of a statement $s \in \mathbf{Stmt}$ is defined by

$$[\![x := Ax + b]\!]X := \{Ax + b \mid x \in X\},$$
$$[\![x^\top Ax + 2b^\top x \leq c]\!]X := \{x \in X \mid x^\top Ax + 2b^\top x \leq c\},$$

for $X \subseteq \mathbb{R}^n$. A program G is a triple $(N, E, \mathbf{st})$, where N is a finite set of *control-points*, $E \subseteq N \times \mathbf{Stmt} \times N$ is the set of control-flow edges, and $\mathbf{st} \in N$ is the start control-point. As usual, the *collecting semantics* V of a program $G = (N, E, \mathbf{st})$ is the least solution of the following constraint system:

$$\mathbf{V}[\mathbf{st}] \supseteq I \qquad \mathbf{V}[v] \supseteq [\![s]\!](\mathbf{V}[u]) \quad \text{for all } (u, s, v) \in E$$

Here, the variables $\mathbf{V}[v]$, $v \in N$ take values in $2^{\mathbb{R}^n}$. The set $I \subseteq \mathbb{R}^n$ is an initial set of states. The components of the collecting semantics V are denoted by $V[v]$ for all $v \in N$.

According to the definitions of Adjé et al. [2], we define *quadratic zones* as follows: A set P of templates $p : \mathbb{R}^n \to \mathbb{R}$ is a *quadratic zone* iff every template $p \in P$ can be written as $p(x) = x^\top A_p x + 2b_p^\top x$, where $A_p \in S\mathbb{R}^{n\times n}$ and $b_p \in \mathbb{R}^n$ for all $p \in P$.

In the following we assume that $P = \{p_1, \dots, p_m\}$ is a finite quadratic zone. We assume w.l.o.g. that $p_i \neq 0$ holds for all $i = 1, \dots, m$. The *abstraction* $\alpha : 2^{\mathbb{R}^n} \to P \to \overline{\mathbb{R}}$ and the *concretization* $\gamma : (P \to \overline{\mathbb{R}}) \to 2^{\mathbb{R}^n}$ are defined as follows:

$$\begin{aligned} \gamma(v) &:= \{x \in \mathbb{R}^n \mid \forall p \in P. p(x) \leq v(p)\}, \quad v : P \to \overline{\mathbb{R}}, \\ \alpha(X) &:= \bigwedge \{v : P \to \overline{\mathbb{R}} \mid \gamma(v) \supseteq X\}, \quad X \subseteq \mathbb{R}^n. \end{aligned}$$

As shown by Adjé et al. [2], α and γ form a Galois-connection. The elements from $\gamma(P \to \overline{\mathbb{R}})$ and the elements from $\alpha(2^{\mathbb{R}^n})$ are called *closed*. $\alpha(\gamma(v))$ is called the *closure of* $v : P \to \mathbb{R}$. Accordingly, $\gamma(\alpha(X))$ is called the *closure of* $X \subseteq \mathbb{R}^n$.

The *abstract semantics* $[\![s]\!]^\sharp : (P \to \overline{\mathbb{R}}) \to P \to \overline{\mathbb{R}}$ of a statement s is defined by $[\![s]\!]^\sharp := \alpha \circ [\![s]\!] \circ \gamma$. The *abstract semantics* $V^\sharp$ of a program $G = (N, E, \mathbf{st})$ is the least solution of the following constraint system:

$$\mathbf{V}^\sharp[\mathbf{st}] \geq \alpha(I) \qquad \mathbf{V}^\sharp[v] \geq [\![s]\!]^\sharp(\mathbf{V}^\sharp[u]) \quad \text{for all } (u, s, v) \in E$$

Here, the variables $\mathbf{V}^\sharp[v]$, $v \in N$ take values in $P \to \overline{\mathbb{R}}$. The components of the abstract semantics $V^\sharp$ are denoted by $V^\sharp[v]$ for all $v \in N$.

The problem of deciding, whether for a given quadratic zone P, a given $v : P \to \overline{\mathbb{Q}}$, a given $p \in P$, and a given $q \in \overline{\mathbb{Q}}$, $\alpha(\gamma(v))(p) \leq q$ holds, is NP-hard (cf. Adjé et al. [2]) and thus intractable. Therefore we use the *relaxed abstract semantics* $V^{\mathcal{R}}$ introduced by Adjé et al. [2] which is based on Shor's semidefinite relaxation schema. However, in order to use our strategy improvement algorithm, we have to switch to the semi-definite dual. This is in fact an advantage, since we gain additional precision through this step.

Definition 5 **($[\![x := Ax + b]\!]^{\mathcal{R}}$).** *For an affine assignment $x := Ax + b$, we define the* relaxed abstract transformer $[\![x := Ax + b]\!]^{\mathcal{R}} : (P \to \overline{\mathbb{R}}) \to P \to \overline{\mathbb{R}}$ *by*

$$[\![x := Ax + b]\!]^{\mathcal{R}} v\,(p) := \sup\{\overline{A}(p) \bullet X \mid \forall p' \in P. \overline{A}_{p'} \bullet X \leq v(p'), X \succeq 0, X_{1\cdot 1} = 1\}$$

for $v : P \to \mathbb{R}$ and $p \in P$, where, for all $p' \in P$,

$$A(p) := A^\top A_p A, \quad b(p) := A^\top A_p b + A^\top b_p, \quad c(p) := b^\top A_p b + 2b_p^\top b$$

$$\overline{A}(p) := \begin{pmatrix} c(p) & b^\top(p) \\ b(p) & A(p) \end{pmatrix}, \quad \overline{A}_{p'} := \begin{pmatrix} 0 & b_{p'}^\top \\ b_{p'} & A_{p'} \end{pmatrix}. \qquad \square$$

Definition 6 **($[\![x^\top Ax + 2b^\top x \leq c]\!]^{\mathcal{R}}$).** *For a quadratic guard $x^\top Ax + 2b^\top x \leq c$, we define the* relaxed abstract transformer $[\![x^\top Ax + 2b^\top x \leq c]\!]^{\mathcal{R}} : (P \to \overline{\mathbb{R}}) \to P \to \overline{\mathbb{R}}$ *by*

$$\begin{aligned} &[\![x^\top Ax + 2b^\top x \leq c]\!]^{\mathcal{R}} v\,(p) \\ := \;& \sup\{\overline{A}_p \bullet X \mid \forall p' \in P. \overline{A}_{p'} \bullet X \leq v(p'), \widetilde{A} \bullet X \leq 0, X \succeq 0, X_{1\cdot 1} = 1\} \end{aligned}$$

for $v : P \to \mathbb{R}$ and $p \in P$, where, for all $p' \in P$,

$$\widetilde{A} := \begin{pmatrix} -c & b^\top \\ b & A \end{pmatrix}, \quad \overline{A}_{p'} := \begin{pmatrix} 0 & b_{p'}^\top \\ b_{p'} & A_{p'} \end{pmatrix}. \qquad \square$$

Our relaxed abstract transformer is the semidefinite dual of the relaxed abstract transformer used by Adjé et al. [2]. Thus, by weak-duality, our relaxation is at least as precise as the relaxation used by Adjé et al. [2]. For all statements s, $[\![s]\!]^{\mathcal{R}}$ is indeed a relaxation of the abstract semantics $[\![s]\!]^\sharp$:

Lemma 9. *The following statements hold for every statement $s \in \mathbf{Stmt}$:*
1. $[\![s]\!]^\sharp \leq [\![s]\!]^{\mathcal{R}}$
2. *For every $i \in \{1, \ldots, m\}$, there exist $\mathcal{A}, a, \mathcal{B}, C$ such that*

$$[\![s]\!]^{\mathcal{R}} v\,(p_i) = \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C}(v(p_1), \ldots, v(p_m))$$

holds for all $v : P \to \overline{\mathbb{R}}$. $\square$

A relaxation of the closure operator is given by $[\![x := x]\!]^{\mathcal{R}}$, i.e., $\alpha \circ \gamma \leq [\![x := x]\!]^{\mathcal{R}}$.

Relaxed Abstract Semantics. The *relaxed abstract semantics* $V^{\mathcal{R}}$ of a program $G = (N, E, \mathbf{st})$ with initial states I is finally defined as the least solution of the following constraint system over $P \to \overline{\mathbb{R}}$:

$$\mathbf{V}^{\mathcal{R}}[\mathbf{st}] \geq \alpha(I) \qquad \mathbf{V}^{\mathcal{R}}[v] \geq [\![s]\!]^{\mathcal{R}}(\mathbf{V}^{\mathcal{R}}[u]) \quad \text{for all } (u, s, v) \in E$$

Here, the variables $\mathbf{V}^{\mathcal{R}}[v]$, $v \in N$ take values in $P \to \overline{\mathbb{R}}$. The components of the relaxed abstract semantics $V^{\mathcal{R}}$ are denoted by $V^{\mathcal{R}}[v]$ for all $v \in N$.

The relaxed abstract semantics is a safe over-approximation of the abstract semantics. Moreover, if all templates and all guards are linear, the relaxed abstract semantics is precise (cf. Adjé et al. [2]):

Lemma 10. $V^\sharp \leq V^{\mathcal{R}}$. *If all templates and all guards are linear, then $V^\sharp = V^{\mathcal{R}}$.* $\square$

Computing Relaxed Abstract Semantics. We now compute the relaxed abstract semantics $V^{\mathcal{R}}$ of a program $G = (N, E, \mathbf{st})$ with initial states I. For that, we define $\mathcal{C}$ to be the constraint system

$$\begin{aligned} \mathbf{x}_{\mathbf{st},p_i} &\geq \alpha(I)(p_i) && \text{for all } i = 1, \ldots, m \\ \mathbf{x}_{v,p_i} &\geq ([\![s]\!]^{\mathcal{R}}(\mathbf{x}_{u,p_1}, \ldots, \mathbf{x}_{u,p_m})^\top)(p_i) && \text{for all } (u, s, v) \in E, i \in \{1, \ldots, m\} \end{aligned}$$

which uses the variables $\mathbf{X} = \{\mathbf{x}_{v,p} \mid v \in N,\ p \in P\}$. The variable $\mathbf{x}_{v,p_i}$ contains the value for the bound for p_i at control-point v.

Because of lemma 9, from $\mathcal{C}$ we can construct a system $\mathcal{E}$ of SDP-equations with $\mu[\![\mathcal{E}]\!] = \mu[\![\mathcal{C}]\!]$ in linear time by introducing auxiliary variables. We have:

Lemma 11. *$V^{\mathcal{R}}[v](p) = \mu[\![\mathcal{E}]\!](\mathbf{x}_{v,p})$ for all $v \in N$ and all $p \in P$.* □

Since $\mathcal{E}$ is a system of SDP-equations, by theorem 2, we can compute the least solution $\mu[\![\mathcal{E}]\!]$ of $\mathcal{E}$ using our strategy improvement algorithm, whenever $\mu[\![\mathcal{E}]\!] \lhd \underline{\infty}$ holds. Thus we have finally shown the following main result:

Theorem 3. *Let $V^{\mathcal{R}}$ be the relaxed abstract semantics of a program $G = (N, E, \mathbf{st})$. Assume that $V^{\mathcal{R}}[v](p) < \infty$ holds for all $v \in N$ and all $p \in P$. We can compute $V^{\mathcal{R}}$ using our strategy improvement algorithm. Each strategy improvement step can by performed by solving $|N| \cdot |P|$ SDP problems, each of which can be constructed in polynomial time. The number of strategy improvement steps is exponentially bounded by the number of merge points in the program G.* □

Example 8. In order to give a complete picture of our method, we now discuss the harmonic oscillator example of Adjé et al. [2] in detail. The program consists only of the simple loop **while** (**true**) $x := Ax$, where $x = (x_1, x_2)^\top \in \mathbb{R}^2$ is the vector of program variables and $A = \begin{pmatrix} 1 & 0.01 \\ -0.01 & 0.99 \end{pmatrix}$.We assume that $I = [0,1] \times [0,1]$ is the set of initial states. The set of control-points just consists of $\mathbf{st}$, i.e. $N = \{\mathbf{st}\}$. The set $P = \{p_1, \ldots, p_5\}$ is given by $p_1(x_1, x_2) = -x_1$, $p_2(x_1, x_2) = x_1$, $p_3(x_1, x_2) = -x_2$, $p_4(x_1, x_2) = x_2$, $p_5(x_1, x_2) = 2x_1^2 + 3x_2^2 + 2x_1x_2$. The abstract semantics is thus finally given by the least solution of the following system of SDP-equations:

$$\begin{aligned}
\mathbf{x}_{\mathbf{st},p_1} &= -\infty \vee 0 \vee \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C_1}(\mathbf{x}_{\mathbf{st},p_1}, \mathbf{x}_{\mathbf{st},p_2}, \mathbf{x}_{\mathbf{st},p_3}, \mathbf{x}_{\mathbf{st},p_4}, \mathbf{x}_{\mathbf{st},p_5}) \\
\mathbf{x}_{\mathbf{st},p_2} &= -\infty \vee 1 \vee \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C_2}(\mathbf{x}_{\mathbf{st},p_1}, \mathbf{x}_{\mathbf{st},p_2}, \mathbf{x}_{\mathbf{st},p_3}, \mathbf{x}_{\mathbf{st},p_4}, \mathbf{x}_{\mathbf{st},p_5}) \\
\mathbf{x}_{\mathbf{st},p_3} &= -\infty \vee 0 \vee \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C_3}(\mathbf{x}_{\mathbf{st},p_1}, \mathbf{x}_{\mathbf{st},p_2}, \mathbf{x}_{\mathbf{st},p_3}, \mathbf{x}_{\mathbf{st},p_4}, \mathbf{x}_{\mathbf{st},p_5}) \\
\mathbf{x}_{\mathbf{st},p_4} &= -\infty \vee 1 \vee \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C_4}(\mathbf{x}_{\mathbf{st},p_1}, \mathbf{x}_{\mathbf{st},p_2}, \mathbf{x}_{\mathbf{st},p_3}, \mathbf{x}_{\mathbf{st},p_4}, \mathbf{x}_{\mathbf{st},p_5}) \\
\mathbf{x}_{\mathbf{st},p_5} &= -\infty \vee 7 \vee \mathbf{SDP}_{\mathcal{A},a,\mathcal{B},C_5}(\mathbf{x}_{\mathbf{st},p_1}, \mathbf{x}_{\mathbf{st},p_2}, \mathbf{x}_{\mathbf{st},p_3}, \mathbf{x}_{\mathbf{st},p_4}, \mathbf{x}_{\mathbf{st},p_5})
\end{aligned}$$

Here

$$\mathcal{A} = \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right) \qquad a = (1)$$

$$\mathcal{B} = \left(\begin{pmatrix} 0 & -0.5 & 0 \\ -0.5 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0.5 & 0 \\ 0.5 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & -0.5 \\ 0 & 0 & 0 \\ -0.5 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0.5 \\ 0 & 0 & 0 \\ 0.5 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 3 \end{pmatrix} \right)$$

$$C_1 = \begin{pmatrix} 0 & -0.5 & -0.005 \\ -0.5 & 0 & 0 \\ -0.005 & 0 & 0 \end{pmatrix} \qquad C_2 = \begin{pmatrix} 0 & 0.5 & 0.005 \\ 0.5 & 0 & 0 \\ 0.005 & 0 & 0 \end{pmatrix}$$

$$C_3 = \begin{pmatrix} 0 & 0.005 & -0.495 \\ 0.005 & 0 & 0 \\ -0.495 & 0 & 0 \end{pmatrix} \qquad C_4 = \begin{pmatrix} 0 & -0.005 & 0.495 \\ -0.005 & 0 & 0 \\ 0.495 & 0 & 0 \end{pmatrix}$$

$$C_5 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1.9803 & 0.9802 \\ 0 & 0.9802 & 2.9603 \end{pmatrix}$$

In this example we have $3^5 = 243$ different $\vee$-strategies. It is clear that the algorithm will switch to the strategy that is given by the finite constants in the first step. At each equation, it then can switch to the **SDP**-expression, but then, because it constructs a strictly increasing sequence, it can never return to the constant. Summarizing, because of the simple structure, it is clear that our strategy improvement algorithm will perform at most 6 strategy improvement steps. In fact our prototypical implementation performs 4 strategy improvement steps on this example. □

6 Experimental Results

A prototypical implementation in OCaml 3.11.0 which uses CSDP 6.0.1 [5] can be downloaded under http://www2.in.tum.de/˜gawlitza/sas10.html. It has been tested under MacOS X 10.5.8 and Debian Linux. We considered the benchmark problems from Adjé et al. [2]. The program oscillator is the implementation of an Euler integration scheme for a harmonic oscillator. The program rotation rotates a higher-dimensional sphere where the analyzer automatically verifies that it is invariant under rotation. The program symplectic is a discretization of the differential equation $\ddot{x} + x = 0$ which starts at a position $x \in [0, 1]$. The program symplecticseu is similar but assumes a threshold $v \geq 0.5$ for the velocity v. For a detailed discussion of the significance of these examples, see Adjé et al. [2]. In all these examples, the least fixpoint is reached after very few iterations:

Example	time (in seconds)	number of improvement steps
filtre	0.16	3
oscillator	0.60	4
rotation	0.04	2
symplectic	0.53	4
symplecticseu	2.82	4

Furthermore the results for symplectic and symplecticseu are more precise than the results obtained by the policy iteration method of Adjé et al. [2]. For symplectic, for instance, we find: $-1.2638 \leq x \leq 1.2638$, and $-1.2653 \leq v \leq 1.2653$. For the other examples we obtain the same results, but considerably faster.

7 Conclusion

We introduced systems of order-concave equations. This class is a natural and strict generalization of systems of rational equations (studied by Gawlitza and Seidl [11, 13]). We showed how the max-strategy improvement approach from Gawlitza and Seidl [11, 12] can be generalized to systems of order-concave equations — provided that the least solution is finite. We thus proved that our algorithm allows to compute the relaxed abstract semantics w.r.t. the abstract domain introduced by Adjé et al. [2]. For future work, we are also interested in studying the use of other convex relaxation schemes in order to deal with more general polynomial templates, a problem already posed by Adjé et al. [2]. This would enable us to analyze not only programs with affine assignments and quadratic guards precisely. Moreover, it remains to investigate further applications.

Acknowledgment. We thank S. Gaubert and D. Monniaux for valuable discussions.

References

[1] Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.): ICALP 2008, Part I. LNCS, vol. 5125, pp. 973–978. Springer, Heidelberg (2008)
[2] Adjé, A., Gaubert, S., Goubault, E.: Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 23–42. Springer, Heidelberg (2010) ISBN 978-3-642-11956-9
[3] Björklund, H., Sandberg, S., Vorobyov, S.: Optimization on completely unimodal hypercubes. Technichal report 2002-18, Department of Information Technology, Uppsala University (2002)
[4] Björklund, H., Sandberg, S., Vorobyov, S.: Complexity of Model Checking by Iterative Improvement: the Pseudo-Boolean Framework. In: Broy, M., Zamulin, A.V. (eds.) PSI 2003. LNCS, vol. 2890, pp. 381–394. Springer, Heidelberg (2004)
[5] Borchers, B.: Csdp, a c library for semidefinite programming. In: Optimization Methods and Software, vol. 11, p. 613. Taylor and Francis, Abington (1999)
[6] Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 462–475. Springer, Heidelberg (2005)
[7] Esparza, J., Gawlitza, T., Kiefer, S., Seidl, H.: Approximative methods for monotone systems of min-max-polynomial equations. In: Aceto et al. [1], pp. 698–710, ISBN 978-3-540-70574-1
[8] Etessami, K., Yannakakis, M.: Recursive concurrent stochastic games. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 324–335. Springer, Heidelberg (2006) ISBN 3-540-35907-9
[9] Etessami, K., Wojtczak, D., Yannakakis, M.: Recursive stochastic games with positive rewards. In: Aceto et al. [1], pp.711–723, ISBN 978-3-540-70574-1
[10] Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static analysis by policy iteration on relational domains. In: Nicola [16], pp. 237–252, ISBN 978-3-540-71314-2
[11] Gawlitza, T., Seidl, H.: Precise relational invariants through strategy iteration. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 23–40. Springer, Heidelberg (2007) ISBN 978-3-540-74914-1
[12] Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: Nicola [16], pp. 300–315 (2007) ISBN 978-3-540-71314-2
[13] Gawlitza, T.M., Seidl, H.: Solving systems of rational equations through strategy iteration. Technical report, TUM (2009)
[14] Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001) ISBN 3-540-42068-1
[15] Miné, A.: The octagon abstract domain. In: WCRE, p. 310(2001)
[16] De Nicola, R. (ed.): ESOP 2007. LNCS, vol. 4421. Springer, Heidelberg (2007) ISBN 978-3-540-71314-2
[17] Ortega, J., Rheinboldt, W.: Iterative solution of nonlinear equations in several variables. Academic Press, London (1970)
[18] Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005) ISBN 3-540-24297-X
[19] Todd, M.J.: Semidefinite optimization. Acta Numerica 10, 515–560 (2001)
[20] Wojtczak, D., Etessami, K.: Premo: An analyzer for probabilistic recursive models. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 66–71. Springer, Heidelberg (2007) ISBN 978-3-540-71208-4

BOXES: A Symbolic Abstract Domain of Boxes

Arie Gurfinkel and Sagar Chaki

Carnegie Mellon University

Abstract. Numeric abstract domains are widely used in program analyses. The simplest numeric domains over-approximate disjunction by an imprecise join, typically yielding path-insensitive analyses. This problem is addressed by domain refinements, such as finite powersets, which provide exact disjunction. However, developing correct and efficient disjunctive refinement is challenging. First, there must be an efficient way to represent and manipulate abstract values. The simple approach of using "sets of base abstract values" is often not scalable. Second, while a widening must strike the right balance between precision and the rate of convergence, it is notoriously hard to get correct. In this paper, we present an implementation of the BOXES abstract domain – a refinement of the well-known BOX (or Intervals) domain with finite disjunctions. An element of BOXES is a finite union of boxes, i.e., expressible as a propositional formula over upper- and lower-bounds constraints. Our implementation is symbolic, and weds the strengths of Binary Decision Diagrams (BDDs) and BOX. The complexity of the operations (meet, join, transfer functions, and widening) is polynomial in the size of the operands. Empirical evaluation indicates that the performance of BOXES is superior to other existing refinements of BOX with comparable expressiveness.

1 Introduction

Numeric abstract domains are widely used in Abstract Interpretation and Software Model-Checking to infer numeric relationships between program variables. To a large extent, the scalability of the most common domains, intervals, octagons, and polyhedra, comes from the fact that they represent convex sets – i.e., conjunctions of linear constraints. This inability to precisely represent disjunction (and disjunctive invariants) is also their main limitation. It means that analyses dependent on such domains are path-insensitive and produce a high-rate of false positives when applied to verification tasks. In practice, an analyzer based on such a domain uses a *disjunctive refinement* to extend the base domain with disjunctions (or a *disjunctive completion* [8] when all disjunctions are added), and thereby increase its precision.

There are several standard ways to build a disjunctive refinement. The simplest one is to allow a *bounded number of disjuncts* (e.g., [15,17,4,11,2,12]). This is typically implemented via finite sets as abstract values (e.g., using $\{a, b\}$ for $a \vee b$), splitting locations in the control flow graph (e.g., unrolling loops, duplicating join points, etc.), or a combination of the two. It has an easy implementation based entirely on the abstract operations (i.e., image, widen, etc.) of the base domain. However, it does not scale to a large number of disjuncts.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 287–303, 2010.
© Springer-Verlag Berlin Heidelberg 2010

The *finite powerset construction* [1,2] represents disjunctions with finite sets and does not bound the number of disjuncts. Most abstract operations easily extend from the base domain. However, it does not scale to a large number of disjuncts, and widening is notoriously hard to get right (see [2] for examples).

If the base domain is finite, like in predicate abstraction [10], Binary Decision Diagrams (BDDs) [5] over the basis (i.e., predicates) of the base domain is the natural choice for the completion. This approach easily scales to a large number of disjuncts, which is particularly important for a "coarse" base domain. Additionally, the canonicity properties of BDDs eliminate redundant abstract values with the same concretization (which are common with the other approaches).

In this paper, we present a new abstract domain, BOXES, that is a disjunctive refinement of the well-known BOX [7] (or Intervals) domain. That is, each value of BOXES is a propositional formula over interval constraints. We make several contributions. *First*, BOXES values are represented by Linear Decision Diagrams [6] (LDDs), a data structure developed by us in prior work. LDDs are an extension of BDDs to formulas over Linear Arithmetic (LA), and are implemented on top of the state-of-the-art BDD package [18]. Thus, BOXES enjoys all of the advantages of a BDD-based disjunctive refinement, including canonicity of representation and scalability to many disjuncts. This is especially important since BOX is very coarse. BOXES is not only more expressive than BOX or a BDD-based domain, but, in practice, can effectively replace either.

Second, we implement algorithms for image computation of transfer functions for BOXES. *Third*, we develop a novel widening algorithm for BOXES, prove its correctness, and implement it via LDDs. Our widening does not fit any of the known widening schemes for disjunctive refinements, and is of independent interest. *Finally*, we evaluate BOXES on an extensive benchmark against state-of-the-art implementations [3] of BOX and its finite powerset.

There has been a significant amount of research on extending BDDs to formulas over LA, especially in the area of timed- and hybrid-verification (e.g., [19,16,13,20]). In contrast, we concentrate on transfer functions common in program analysis applications; to our knowledge, we are the first to consider widening in this context; and, the first to conduct extensive evaluation of such an approach to a program analysis task.

The rest of the paper is structured as follows. Section 2 gives a brief overview of LDDs. Section 3 presents our abstract domain BOXES. Section 4 describes widening. Section 5 compares BOXES with the finite powerset of BOX. Section 6 presents our experimental results, and Section 7 concludes.

2 Linear Decision Diagrams

In this section, we briefly review Linear Decision Diagrams (LDDs) that are the basis for our abstract domain. For more details see [6].

A decision diagram (DD) is a directed acyclic graph (DAG) in which non-terminal nodes are labeled with decisions and terminal nodes are labeled with values. LDD is a DD with non-terminal nodes labeled by LA constraints and two terminal nodes representing TRUE and FALSE, respectively. LDDs are a natural representation for propositional formulas over LA.

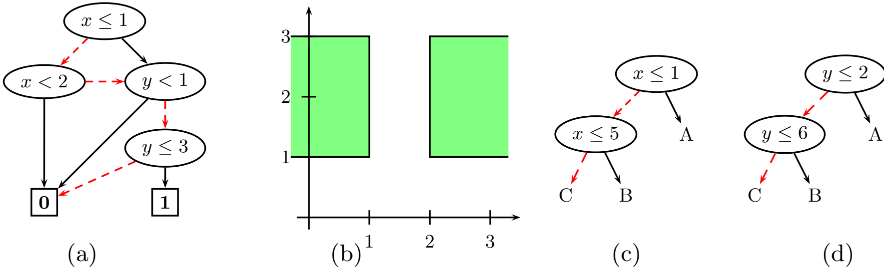


Fig. 1. (a) An LDD and (b) its geometric interpretation; (c), (d) two LDDs

Example 1. An example of an LDD $(x \leq 1 \vee x \geq 2) \wedge (1 \leq y \leq 3)$ and its geometric interpretation are shown in Fig. 1(a) and Fig. 1(b). Oval and boxed nodes represent non-terminal and terminal nodes, respectively. Solid and dashed edges represent high (TRUE) and low (FALSE) branches, respectively. □

Formally, an LDD over a fragment T of LA is a DAG with

– Two terminal nodes labeled by **0** and **1**, respectively.
– Non-terminal nodes. Each non-terminal node u has two children, denoted by $H(u)$ and $L(u)$, and is labeled with a T-atom (i.e., an atomic predicate), denoted by $C(u)$.
– Edges, high $(u, H(u))$ and low $(u, L(u))$, for every non-terminal node u.

We write $attr(u)$ for $(C(u), H(u), L(u))$.

An LDD with a root node u represents the formula $\exp(u)$ over T defined by:

$$\exp(u) \triangleq \begin{cases} \text{FALSE} & \text{if } u = \mathbf{0} \\ \text{TRUE} & \text{if } u = \mathbf{1} \\ \text{ITE}(C(u), \exp(H(u)), \exp(L(u))) & \text{otherwise}, \end{cases} \quad (1)$$

where $\text{ITE}(a, b, c) \triangleq (a \wedge b) \vee (\neg a \wedge c)$.

For simplicity, we do not distinguish between a node u and $\exp(u)$.

Let V be a set of variables. We write $\mathbb{UBQ}$ for the set $\{x \lesssim k \mid x \in V, k \in \mathbb{Q}, \lesssim \in \{<, \leq\}\}$ of rational[1] upper bound constraints, and $\mathbb{IVQ}$ for the set $\{x \sim k \mid x \in V, k \in \mathbb{Q}, \sim \in \{<, \leq, =, \geq, >\}\}$ of rational interval constraints. In this paper, we restrict LDDs to $\mathbb{UBQ}$. This is sufficient to represent any propositional formula over $\mathbb{IVQ}$. For example, $x \leq 5$ and $x > 5$ correspond to LDDs $\text{ITE}(x \leq 5, \mathbf{1}, \mathbf{0})$, and $\text{ITE}(x \leq 5, \mathbf{0}, \mathbf{1})$, respectively. For $(x \sim k) \in \mathbb{IVQ}$, we write $\text{VAR}(x \sim k)$ for x.

Let $\preceq \subseteq V \times V$ be a total order on V. We extend it to $\mathbb{UBQ}$ in a natural way:

$$(x_1 \lesssim_1 k_1) \preceq (x_2 \lesssim_2 k_2) \text{ iff } (x_1 \preceq x_2) \vee ((x_1 \lesssim_1 k_1) \Rightarrow (x_2 \lesssim_2 k_2)), \quad (2)$$

and to LDD nodes:

$$u \preceq v \text{ iff } (v \in \{\mathbf{0}, \mathbf{1}\}) \vee (u \notin \{\mathbf{0}, \mathbf{1}\} \wedge C(u) \preceq C(v)). \quad (3)$$

[1] While we use rationals for ease of presentation, our results extend to integers.

Table 1. Basic LDD operations. U is a set of variables.

Operation	Semantics	Complexity	Operation	Semantics	Complexity
$\text{AND}(f, g)$	$f \wedge g$	$O(\|f\| \cdot \|g\|)$	$\text{OR}(f, g)$	$f \vee g$	$O(\|f\| \cdot \|g\|)$
$\text{ITE}(h, f, g)$	$(h \wedge f) \vee (\neg h \wedge g)$	$O(\|h\| \cdot \|f\| \cdot \|g\|)$	$\text{LEQ}(f, g)$	$f \Rightarrow g$	$\|f\| \cdot \|g\|$
$\text{NOT}(f)$	$\neg f$	$O(\|f\|)$	$\text{EXIST}(U, f)$	$\exists U \cdot f$	$O(\|f\| \cdot 2^{\|U\|})$

```
1: function LEQ (LDD f, LDD g)
2:     if (f = g) ∨ (f = 0) ∨ (g = 1) then
3:         return TRUE
4:     if (f = 1) ∨ (g = 0) then
5:         return FALSE
6:     if C(f) ⪯ C(g) then v ← C(f)
7:     else  v ← C(g)
8:     return LEQ(f|_v, g|_v) ∧ LEQ(f|_¬v, g|_¬v)
```

```
Require: FNPOS and FNNEG map constraints into LDDs
1: function RC(var x, LDD f, fun FNPOS, fun FNNEG)
2:     if (f = 0) ∨ (f = 1) then return f
3:     v ← C(f)
4:     t ← RC(x, f|_v, FNPOS, FNNEG)
5:     e ← RC(x, f|_¬v, FNPOS, FNNEG)
6:     if x ≠ VAR(v) then return ITE(v, t, e)
7:     t ← AND(FNPOS(v), t)
8:     e ← AND(FNNEG(v), e)
9:     return OR(t, e)
```

Fig. 2. LDD algorithms: LEQ – decides implication, and RC replaces constraints

An LDD u is *ordered* w.r.t. $\preceq$ iff for every node v reachable from u, $v \preceq H(v)$ and $v \preceq L(v)$. An LDD over $\mathbb{UBQ}$ is *locally reduced* iff the following five conditions hold on every internal node u and v: (1) *No duplicate nodes.* $attr(u) = attr(v) \Rightarrow u = v$; (2) *No redundant nodes.* $L(v) \neq H(v)$; (3) *Normalized labels.* $C(v) \in \mathbb{UBQ}$; (4) *Imply high.* $\neg(C(v) \Rightarrow C(H(v)))$; (5) *Imply low.* If $C(v) \Rightarrow C(L(v))$ then $H(v) \neq H(L(v))$.

For a fixed variable order, ROLDDs are canonical for propositional formulas over $\mathbb{IVQ}$. Specifically, if u and v represent semantically equivalent expressions $(\exp(u) \Leftrightarrow \exp(v))$, then $u = v$. From here on, we fix a total order $\preceq$ on V and say LDD to mean Reduced Ordered LDDs (ROLDD).

Like BDDs, LDDs provide polynomial time algorithms for the basic propositional operations: disjunction (union), conjunction (intersection), negation (complement), and existential quantification (projection). These are summarized in Table 1. Like BDDs, in the worst case, the size of an LDD is exponential in the number of variables. In some implementations (e.g., in [6]) negation ($\neg f$) is a constant time operation. For completeness, the pseudo code for LEQ is shown in Fig. 2. This, and all other DD algorithms in this paper, are implicitly memoized – results of all intermediate operations are cached and reused as needed. For an LDD f (or its corresponding ITE-expression) and a constraint $v \in \mathbb{UBQ}$, we write $f|_v$ and $f|_{\neg v}$ for, respectively, the positive and the negative cofactor of f w.r.t. v. Let $f[u/w]$ denote the result of the substitution of constraint w for every occurrence of constraint u in f. Then, the cofactors are defined as follows:

$$f|_v \triangleq f[u/\text{TRUE} \mid v \Rightarrow u] \qquad f|_{\neg v} \triangleq f[u/\text{FALSE} \mid u \Rightarrow v]\,. \tag{4}$$

In this paper, we do not distinguish between propositional formulas over $\mathbb{IVQ}$ the corresponding LDDs. For example, we write $f \wedge (1 \leq x \leq 10)$ to mean an LDD obtained by conjunction of an LDD for f and an LDD for $1 \leq x \leq 10$.

3 The Boxes Abstract Domain

Let $\mathbb{R}^n$ be an n-dimensional real vector space. A set $\mathcal{B} \subseteq \mathbb{R}^n$ is a *rational box* iff it is expressible by a finite system of rational interval constraints. The set of all rational boxes of $\mathbb{R}^n$ is denoted by $\mathbb{B}^n$. The Box abstract domain [7] is a tuple $(\mathbb{B}^n, \subseteq, \emptyset, \mathbb{R}^n, \uplus, \cap)$, where $\subseteq$ is the subset ordering, $\emptyset$ is the empty set, $\uplus$ is the box hull (i.e., for any two boxes $\mathcal{B}_1$ and $\mathcal{B}_2$, $\mathcal{B}_1 \uplus \mathcal{B}_2 = \mathcal{B}_3$ is the smallest rational box s.t. $\mathcal{B}_1 \cup \mathcal{B}_2 \subseteq \mathcal{B}_3$), and $\cap$ is set intersection. Note that since $\mathbb{B}^n$ is not closed under union, union is over-approximated by the box hull.

A set $\mathcal{BS} \subseteq \mathbb{R}^n$ is a *set of rational boxes* iff there exist rational boxes $\mathcal{B}_1, \ldots, \mathcal{B}_k$ such that $\mathcal{BS} = \bigcup_{i=1}^{k} \mathcal{B}_i$. The set of all sets of rational boxes of $\mathbb{R}^n$ is denoted by $\mathbb{BS}^n$. The Boxes abstract domain is a tuple $(\mathbb{BS}^n, \subseteq, \emptyset, \mathbb{R}^n, \cup, \cap)$, where $\subseteq$ is the subset ordering, $\emptyset$ is the empty set, and $\cup$ and $\cap$ are set union and intersection, respectively. Boxes abstract domain is a disjunctive refinement of Box domain. Since $\mathbb{BS}^n$ is closed under union, intersection, (and complement) all basic operations are exact. In the rest of this section, we describe our implementation of Boxes using LDDs.

Representation and basic operations. Let $V = \{x_1, \ldots, x_n\}$ be n variables, and $\preceq$ be some total order on V. We assume that each variable is bound to a unique dimension. We use $\boldsymbol{x}$ to denote an element of $\mathbb{R}^n$. Implicitly, $\boldsymbol{x}$ is also a valuation of V, where $\boldsymbol{x}(i)$ is the value of the variable bound to the ith dimension. Then, there is a one-to-one correspondence between Reduced $\preceq$-Ordered LDDs over V and rational boxes over $\mathbb{R}^n$ – each $\mathcal{BS} \in \mathbb{BS}^n$ corresponds to the unique LDD f such that $\boldsymbol{x} \in \mathcal{BS} \Leftrightarrow \boldsymbol{x} \models \exp(f)$. Thus, the domain Boxes is implemented by a tuple $(\mathrm{LDD}(V), \textsc{leq}, \mathbf{0}, \mathbf{1}, \textsc{or}, \textsc{and})$, where $\mathrm{LDD}(V)$ is the set of all LDDs over V. All of the operations are linear in the size of their operands (see Table 1). Note, however, that the size of an LDD over V is in the worst case exponential in $|V|$. In the rest of this paper, we do not distinguish between a set of boxes $\mathcal{BS} \subseteq \mathbb{R}^n$, a corresponding LDD, and a corresponding propositional formula.

In addition to the base operations described above, static analysis applications typically require operations to check for equality and satisfiability (non-emptiness), to compute set-theoretic difference, projection (or unconstraining), images of assignments and guards, and widening (e.g., see [9]). Except for the last two, the operations follow easily from the existing LDD operations. Image computation requires new (but simple) operations, and widening is the most complex one. In the rest of the section, we summarize implementations of the basic and image operations. Widening is deferred to Section 4.

Basic operations. LDDs are canonical for $\mathbb{BS}^n$, hence equality is a constant time operation – two LDD nodes are equivalent iff they have the same attributes. Similarly, satisfiability (and universality) are checked by comparing an LDD to $\mathbf{0}$ (and $\mathbf{1}$). Sometimes (e.g., [2]) it is useful to compute an over-approximation of a set-theoretic difference of two abstract values. Boxes domain is closed under complement and intersection, hence set-theoretic difference is computed exactly using the equivalence: $\mathcal{BS}_1 \setminus \mathcal{BS}_2 \triangleq \mathcal{BS}_1 \cap \neg\mathcal{BS}_2$, where $\mathcal{BS}_1, \mathcal{BS}_2 \in \mathbb{BS}^n$. Note

that set-complement is also computed exactly via LDD negation. Projection of a variable x is done via existential quantification EXIST (see last row of Table 1).

Guards and Assignments. Let c be an assignment or a guard. We write $||c||$ for the concrete semantics of c as a function from $\mathbb{R}^n$ to $\mathbb{R}^n$. For example,

$$||x_i \leq 4||(\mathcal{BS}) = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{BS} \wedge \boldsymbol{x}(i) \leq 4\} .$$

We write $|| \cdot ||_{\mathbb{BS}}$: LDD $\rightarrow$ LDD for an abstract transformer that over-approximates $||\cdot||$ in $\mathbb{BS}^n$. That is, given an LDD f for a set of boxes $\mathcal{BS}$, $||c||_{\mathbb{BS}}(f)$ is an LDD representing the smallest set of boxes over-approximating $||c||(\mathcal{BS})$.

The simplest guard is $k_1 \lesssim_1 x_i \lesssim_2 k_2$. The corresponding abstract transformer just adds the appropriate constraint:

$$||k_1 \lesssim_1 x_i \lesssim_2 k_2||_{\mathbb{BS}}(f) \triangleq f \wedge k_1 \lesssim_1 x_i \lesssim_2 k_2 , \tag{5}$$

where $k_1, k_2 \in \mathbb{Q}$ and $\lesssim_1, \lesssim_2 \in \{<, \leq\}$. Either the lower bound (k_1) or the upper bound (k_2) can be omitted. The simplest assignment is $x_i \leftarrow v$ where v is a symbolic constant such that $k_1 \lesssim_1 v \lesssim_2 k_2$. The abstract transformer is constructed by projecting away the current value of x_i and assuming that x_i is in the same interval as v:

$$||x_i \leftarrow v||_{\mathbb{BS}}(f) \triangleq ||k_1 \lesssim_1 x_i \lesssim_2 k_2||_{\mathbb{BS}}(\exists x_i \cdot f) , \tag{6}$$

where k_1, k_2, $\lesssim_1$, and $\lesssim_2$ are as above.

For the next class of transformers, we introduce the function RC shown in Fig. 2. RC takes a variable x, an LDD f, and two functions FNPOS and FNNEG that map constraints to LDDs, and returns an LDD obtained by replacing every constraint u on x in f by FNPOS(u) on the high-branch of u and by FNNEG(u) on the low branch of u.

For example, let ID $\triangleq \lambda u \cdot u$ and COMP $\triangleq \lambda u \cdot \neg u$ be the identity and the complement functions, respectively. Then, RC(x, f, ID, COMP) is an identity function – every constraint on x is replaced by itself.

Transfer functions implemented with RC are shown in Table 2, where the columns are: 1st – the command, 2nd – assumptions made, 3rd – the implementation as a call to RC, and 4th and 5th – FNPOS and FNNEG functions used by RC, respectively. Throughout, we assume that $a, a_1, a_2, k_1, k_2 \in \mathbb{Q}$, and x and y are two distinct variables. Furthermore, for the guard (the last row), we require that constraints on x precede those on y in the diagram ordering. This is not a limitation – any guard can be rewritten to satisfy this restriction.

The transfer function for $x \leftarrow x + a \cdot y$ is implemented with XPY (see Fig. 3):

$$||x \leftarrow x + a \cdot y||_{\mathbb{BS}}(f) \triangleq \text{XPY}(f, x, a, y) . \tag{7}$$

The intuition behind XPY is to traverse the DD and reduce the transfer function to a simpler one as soon as a bound for either x or y is found. There are two cases based on whether $x \preceq y$ or $y \preceq x$.

Example 2. As a simple example, consider applying the transfer function to LDD f shown in Fig. 1(c). Here, we assume that $x \preceq y$, and A, B, and C are sub-diagrams that do not contain x. Note that f is equivalent to:

$$(x \leq 1 \wedge A) \vee (1 < x \leq 5 \wedge B) \vee (5 < x \wedge C) \tag{8}$$

Table 2. Abstract transformers; v is a symbolic constant bounded by $k_1 \lesssim v \lesssim k_2$, t_1 is $a_1 \cdot x + k_1$, and t_2 is $a_2 \cdot x + k_2$

Action	Assume	Implementation	$\textsc{fnPos}(z \lesssim b)$	$\textsc{fnNeg}(z \lesssim b)$
$x \leftarrow x + v$		$\text{RC}(x, f, \textsc{fnPos}, \textsc{fnNeg})$	$x \lesssim b + k_2$	$\neg(x \lesssim b + k_1)$
$x \leftarrow a \cdot x$	$a > 0$	$\text{RC}(x, f, \textsc{fnPos}, \textsc{fnNeg})$	$x \lesssim a \cdot b$	$\neg(x \lesssim a \cdot b)$
$x \leftarrow a \cdot x$	$a < 0$	$\text{RC}(x, f, \textsc{fnPos}, \textsc{fnNeg})$	$a \cdot b \lesssim x$	$\neg(a \cdot b \lesssim x)$
$x \leftarrow a \cdot y$	$a > 0$	$\text{RC}(y, \exists x \cdot f, \textsc{fnPos}, \textsc{fnNeg})$	$(z \lesssim b) \wedge (x \lesssim a \cdot b)$	$\neg(z \lesssim b) \wedge \neg(x \lesssim a \cdot b)$
$x \leftarrow a \cdot y$	$a < 0$	$\text{RC}(y, \exists x \cdot f, \textsc{fnPos}, \textsc{fnNeg})$	$(z \lesssim b) \wedge (a \cdot b \lesssim x)$	$\neg(z \lesssim b) \wedge \neg(a \cdot b \lesssim x)$
$a_1 \cdot x + a_2 \cdot y \lesssim k$	$a_1 > 0$	**let** $g = \text{RC}(x, f, \textsc{fnPos}, \textsc{fnNeg})$ **in**	$(z \lesssim b)$	$\neg(z \lesssim b) \wedge (a_2 \cdot y \lesssim k - a_1 \cdot b)$
	$a_2 < 0$	$\text{RC}(y, g, \textsc{fnPos}, \textsc{fnNeg})$	$(z \lesssim b) \wedge (a_1 \cdot x \lesssim k - a_2 \cdot b)$	$\neg(z \lesssim b)$

The transformer distributes over disjunction. First, compute $x \leftarrow a \cdot y$ on the sub-diagrams A, B, and C, to get:

$$A' = ||x \leftarrow a \cdot y||(A) \qquad B' = ||x \leftarrow a \cdot y||(B) \qquad C' = ||x \leftarrow a \cdot y||(C)\,. \tag{9}$$

Second, update the results to reflect the bounds on x in A, B, and C:

$$||x \leftarrow x + v_a||(A') \vee ||x \leftarrow x + v_b||(B') \vee ||x \leftarrow x + v_c||(C')\,, \tag{10}$$

where $v_a \leq 1$, $1 < v_b \leq 5$, and $5 < v_c$.

Alternatively, lets apply the same transformer to LDD g shown in Fig. 1(d). Here, we assume $y \preceq x$, and A, B, and C are sub-diagrams that do not contain y. g is equivalent to:

$$(y \leq 2 \wedge A) \vee (2 < y \leq 6 \wedge B) \vee (6 < y \wedge C) \tag{11}$$

In each disjunct the value of y is known:

$$A' = ||x \leftarrow x + v_a||(A) \quad B' = ||x \leftarrow x + v_b||(B) \quad C' = ||x \leftarrow x + v_c||(C)\,,$$

where $v_a \leq 2 \cdot a$, $2 \cdot a < v_b \leq 6 \cdot a$, and $6 \cdot a < v_6$. The final result is: $\text{ITE}(y \leq 2, A', \text{ITE}(y \leq 6, B', C'))$. □

Finally, an abstract transformer for a linear assignment $x \leftarrow a_1 \cdot x_1 + \cdots + a_n \cdot x_n + v$ is computed as a sequence of simpler transformers. For example $||x \leftarrow a \cdot y + b \cdot z + k||_{\mathbb{BS}}$ is reduced to

$$||x \leftarrow k||_{\mathbb{BS}} \circ ||x \leftarrow x + a \cdot y||_{\mathbb{BS}} \circ ||x \leftarrow x + b \cdot z||_{\mathbb{BS}}\,. \tag{12}$$

Theorem 1. *Let c be an action of the form above, and f be the LDD corresponding to $\mathcal{BS} = \{\mathcal{B}_1, \ldots, \mathcal{B}_k\}$. Then, $||c||_{\mathbb{BS}}(f)$ is equivalent to $\cup_{i=1}^{k} ||c||_{\mathbb{B}}(\mathcal{B}_i)$, where $||\cdot||_{\mathbb{B}} : \mathbb{B}^n \to \mathbb{B}^n$ is the abstract transformer of* BOX.

Proof. (Sketch) For simplest transformers, the result follows trivially. The rest are equivalent to applying $||\cdot||_{\mathbb{B}}$ to each **1**-path of f. □

Join and Box Hull of Boxes. Figure 4 presents algorithms for two other operations on LDDs. BOXJOIN(f, g) returns $f \uplus g$, while BOXHULL(f, g) returns the box hull of f and g. BOXHULL invokes BOXJOIN as a subroutine. The complexity of both algorithms is in $O(|f| \cdot |g|)$.

```
Require: x ≠ y
 1: function XPY(LDD f, var x, ℚ a, var y)
 2:     if x ⪯ y then return XPY1(f, x, a, y, TRUE)
 3:     else   return XPY2(f, x, a, y, TRUE)

 4: function XPY1(LDD f, var x, ℚ a, var y, cons c)
 5:     if f = 0 ∨ f = 1 then return f
 6:     u ← C(f)
 7:     if x ⪯ VAR(u) then
 8:         if c = TRUE then return f
 9:         return ||x ← a · y + v||_BS(f), where v ⊨ c
10:     if x ≠ VAR(u) then
11:         t ← XPY1(f, x, a, y, c)
12:         e ← XPY1(f, x, a, y, c)
13:         return ITE(u, t, e)
14:     Assert u is x ≲ b
15:     t ← ||x ← a · y||_BS(f|_u)
16:     t ← ||x ← x + v||_BS(t), where v ⊨ (c ∧ v ≲ b)
17:     e ← XPY1(f|_¬u, x, a, y, ¬(u[x/v]))
18:     return OR(t, e)

19: function XPY2(LDD f, var x, ℚ a, var y, cons c)
20:     if f = 0 ∨ f = 1 then return f
21:     u ← C(f)
22:     if y ⪯ VAR(u) then
23:         if c = TRUE then return EXIST(x, f)
24:         return ||x ← x + v||_BS(f), where v ⊨ c
25:     if y ≠ VAR(u) then
26:         t ← XPY2(f, x, a, y, c)
27:         e ← XPY2(f, x, a, y, c)
28:         return ITE(u, t, e)
29:     Assert u is y ≲ b
30:     t ← ||x ← x + v||_BS(f|_u), where v ⊨ (c ∧ v ≲ a · b)
31:     e ← XPY2(f|_¬u, x, a, y, ¬(v ≲ a · b))
32:     return ITE(u, t, e)
```

Fig. 3. Algorithm to compute abstract transfer function for $x \leftarrow x + a \cdot y$

```
 1: function BOXHULL (LDD f)
 2:     if (f = 0) ∨ (f = 1) then
 3:         return f
 4:     fh ← BOXHULL(H(f))
 5:     fl ← BOXHULL(L(f))
 6:     if L(f) = 0 then
 7:         return ITE(C(f), fh, 0)
 8:     if H(f) = 0 then
 9:         return ITE(C(f), 0, fl))
10:     return BOXJOIN(C(f) ∧ fh, fl)

Require: f and g are singletons.
 1: function BOXJOIN (LDD f, LDD g)
 2:     if (f = g) ∨ (f ∈ {0, 1}) ∨ (g ∈ {0, 1}) then
 3:         return OR(f, g)
 4:     if C(g) ⪯ C(f) then return BOXJOIN(g, f)
 5:     u ← C(f)
 6:     if (f|_u = 0) ∧ (g|_u = 0) then
 7:         return ITE(u, 0, BOXJOIN(L(f), L(g)))
 8:     if (f|_¬u = 0) ∧ (g|_¬u = 0) then
 9:         return ITE(C(g), BOXJOIN(H(f), H(g)), 0)
10:     return BOXJOIN(f|_u = 0 ? L(f) : H(f), g)
```

Fig. 4. Algorithms to compute a box hull and a box join ($\uplus$) of LDDs

4 Widening

In static analysis, widening is used to ensure that the analysis always terminates, even in the presence of infinite ascending chains in the underlying abstract domain [9]. Let $\hat{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an abstract domain. An operation $\nabla_d : D \times D \to D$ is a *widening* for $\hat{D}$ iff it satisfies two conditions: (1) *over-approximation*: $x \sqsubseteq y \Rightarrow (x \sqcup y) \sqsubseteq (x \nabla_d y)$, and (2) *stabilization*: for every increasing sequence $x_1 \sqsubseteq x_2 \cdots$, the (widening) sequence

$$y_1 = x_1\,, \qquad y_i = y_{i-1} \nabla_d (y_{i-1} \sqcup x_i)\,, \text{ for } i > 1 \tag{13}$$

stabilizes, i.e., $\exists k \cdot y_k = y_{k+1}$.

In this section, we describe a widening for BOXES. We proceed in stages. First, we introduce a new domain construction $\text{STEP}(\hat{D})$, called *step (function)* construction, that lifts a domain $\hat{D}$ to (step) functions from $\mathbb{R}$ to D. Second, we give a procedure to lift a widening ∇_d of $\hat{D}$ to a widening ∇_s of $\text{STEP}(\hat{D})$. Finally, we show that n-dimensional BOXES are step constructions of $(n-1)$-dimensional BOXES and implement ∇_s on top of LDDs.

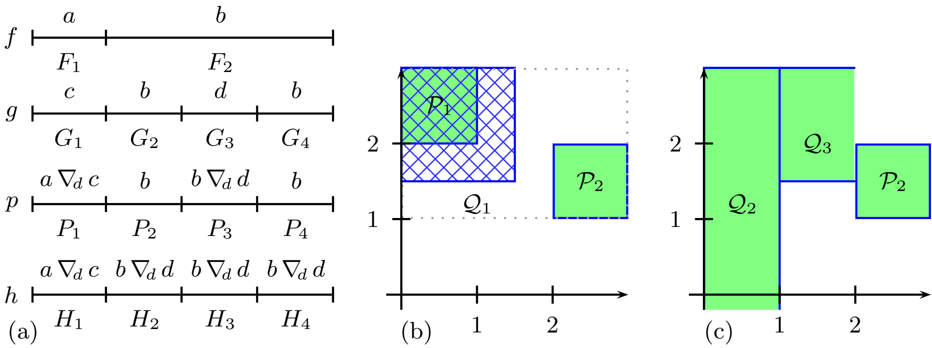


Fig. 5. Example of a widening

Step function construction. A function $f : \mathbb{R} \to D$ is a *step function* if it can be written as a finite combination of intervals. That is,

$$f(x) = (v_1 \sqcap f_1(x)) \sqcup \cdots \sqcup (v_n \sqcap f_n(x)) , \tag{14}$$

where $v_i \in D$, and there exists a partitioning $F_1, \ldots, F_n$ of $\mathbb{R}$ by intervals such that $f_i(x) = \top$ if $x \in F_i$ and $f_i(x) = \bot$ otherwise. A step function f induces an equivalence relation $\equiv_f$ on R:

$$x \equiv_f y \Leftrightarrow \forall z \cdot ((x \leq z \leq y) \vee (y \leq z \leq x)) \Rightarrow f(x) = f(z) . \tag{15}$$

We write $[x]_f$ for the equivalence class of x w.r.t. $\equiv_f$. Note that the index of $\equiv_f$ is finite and the equivalence classes are naturally ordered: $[x] \leq [y] \Leftrightarrow x \leq y$. We assume the classes are enumerated, so the $\leq$-least equivalence class is *first*, the next one is *second*, etc. For step functions f, g, we write $\equiv_{f,g}$ for the relation:

$$x \equiv_{f,g} y \quad \Leftrightarrow \quad (x \equiv_f y) \wedge (x \equiv_g y) , \tag{16}$$

and $[x]_{f,g}$ for the corresponding equivalence class of x.

The set of all step functions from $\mathbb{R}$ to a domain $\hat{D}$, denoted $\mathbb{R} \to_s D$, forms an abstract domain $\text{STEP}(\hat{D}) \triangleq (\mathbb{R} \to_s D, \dot{\sqsubseteq}, \dot{\bot}, \dot{\top}, \dot{\sqcup}, \dot{\sqcap})$. The dot above an operator denotes pointwise extension, e.g., $f \mathrel{\dot{\sqsubseteq}} g \triangleq \forall x \in \mathbb{R} \cdot f(x) \sqsubseteq g(x)$.

One-dimensional BOXES is STEP({TRUE, FALSE}) – the step construction applied to the Boolean domain. Similarly, n-dimensional BOXES is a step construction of $(n-1)$-dimensional BOXES. Since the Boolean domain is finite – it's join and widening coincide. Thus, to get a widening for BOXES, we only need to show how to lift a widening from a base domain to its step construction. We use 1- and 2-dimensional BOXES for examples in the rest of this section.

Lifting widening to $\text{STEP}(\hat{D})$. Clearly, the pointwise extension $\dot{\nabla}_d$, of the widening ∇_d of $\hat{D}$, is not a widening of $\text{STEP}(\hat{D})$. As a counterexample, the divergent sequence $\{(0 \leq x \leq i)\}_{i=1}^{\infty}$ of BOXES values is it's own pointwise widening sequence. Let us examine this in more detail.

Example 3. Let f and g be step functions as illustrated in Fig. 5(a). Each function is shown as a partitioning of the number line with the value above and the name below the line, respectively. Thus, f has two partitions F_1 and F_2 with

values a, and b, respectively. We assume that lower case letters represent distinct elements of some domain $\hat{D}$, ordered alphabetically. Note that $G_1 = F_1$ and F_2 is refined by G_2, G_3, and G_4. Consider $p = f \mathbin{\dot{\nabla}_d} g$ as shown in Fig. 5(a) and compare to f. Clearly, p is on a divergent path. In it, partition F_2 is split into three parts, but both P_2 and P_4 have the same value as F_2. Thus, they can be refined again. A way to ensure convergence is to assign to P_2 and P_4 the value of P_3, as shown by h in Fig. 5(a). This is the intuition for our approach. □

In summary, pointwise widening $f \mathbin{\dot{\nabla}_d} g$ diverges whenever it refines a partition in f without updating its value. Thus, to guarantee convergence, we assign to the offending partition a value of its neighbor that refines the same partition of f. The formal definition is given below:

Definition 1. *Let $\hat{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an abstract domain with a widening ∇_d. Let $f, g \in \mathbb{R} \rightarrow_s D$ be two step functions s.t. $f \mathrel{\dot{\sqsubseteq}} g$, and $[y_1], \ldots, [y_n]$ be the equivalence classes of $\equiv_{f,g}$ enumerated by their natural order $\leq$. Then, the* step widening*, ∇_s, for* $\mathrm{STEP}(\hat{D})$ *is defined as follows:*

$$(f \mathbin{\nabla_s} g)(x) \triangleq \bigsqcup_{i=1}^{n} (v_i \sqcap h_i(x)) \,, \tag{17}$$

where n is the index of $\equiv_{f,g}$, $h_i(x) = \top$ if $x \in [y_i]$ and $h_i(x) = \bot$ otherwise, and

$$v_i \triangleq \begin{cases} f(y_i) \;\nabla_d\; g(y_i) & \text{if } f(y_i) \neq g(y_i) \text{ or } [y_i]_{f,g} = [y_i]_f \\ f(y_i) \;\nabla_d\; g(y_{i+1}) & \text{else if } i < n \text{ and } [y_{i+1}]_{f,g} \subseteq [y_i]_f \\ f(y_i) \;\nabla_d\; g(y_{i-1}) & \text{otherwise} \end{cases} \tag{18}$$

Example 4. Consider two sets of boxes $\mathcal{BS}_1 = \{\mathcal{P}_1, \mathcal{P}_2\}$ and $\mathcal{BS}_2 = \{\mathcal{Q}_1, \mathcal{P}_2\}$ shown in Fig 5(b), where

$$\mathcal{P}_1 = (0 \leq x \leq 1) \wedge (2 \leq y \leq 3) \qquad \mathcal{P}_2 = (2 \leq x \leq 3) \wedge (1 \leq y \leq 2)$$
$$\mathcal{Q}_1 = (0 \leq x \leq 1.5) \wedge (1.5 \leq y \leq 3) \,.$$

The result of $\mathcal{BS}_1 \mathbin{\nabla_s} \mathcal{BS}_2$ is shown in Fig. 5(c). Note that even though $\mathcal{BS}_1$ and $\mathcal{BS}_2$ have the same box hull (shown by a doted frame), their widening is larger. This shows that widening makes it very hard to analytically compare difference in precision between Boxes and Box. □

Theorem 2. *The operator ∇_s defined in Def. 1 is a widening on* $\mathrm{STEP}(\hat{D})$.

Proof. Over-approximation. Let $h = f \mathbin{\nabla_s} g$. We show that for any $i \in [1, n]$, $h(y_i) \sqsupseteq g(y_i)$. Based on (18) there are 3 cases. In case 1, $h(y_i) = f(y_i) \mathbin{\nabla_d} g(y_i) \sqsupseteq g(y_i)$. In case 2, $[y_{i+1}]_{f,g} \subseteq [y_i]_f \Rightarrow f(y_i) = f(y_{i+1})$. Also, $f \mathrel{\dot{\sqsubseteq}} g \Rightarrow f(y_i) = f(y_{i+1}) \sqsubseteq g(y_{i+1})$. Finally, $h(y_i) = f(y_i) \mathbin{\nabla_d} g(y_{i+1}) \sqsupseteq f(y_i) = g(y_i)$. In case 3, we have $[y_{i-1}]_{f,g} \subseteq [y_i]_f$, from which the results follows as in case 2..

Stabilization. Let $f : \mathbb{R} \rightarrow_s D$ be a step function, and $\{f_i\}_{i=1}^{\infty}$ be an infinite sequence defined as follows:

$$f_1 \triangleq f \,, \qquad\qquad f_i \triangleq f_{i-1} \mathbin{\nabla_s} g_i \,, \text{ for } i > 1 \,, \tag{19}$$

where $\{g_i\}_{i=1}^{\infty}$ is any sequence of step function such that $f_{i-1} \dot{\sqsubseteq} g_i$. We show that the sequence stabilizes, i.e., for some k, $f_k \doteq f_{k+1}$.

We write $\equiv_i$ for $\equiv_{f_i}$ and $[x]_i$ for $[x]_{f_i}$. For $i \geq 1$, let $\equiv_{\leq i}$ be the equivalence relation: $x \equiv_{\leq i} y \Leftrightarrow \forall 1 \leq j \leq i \cdot x \equiv_j y$, and $[\cdot]_{\leq i}$ be its equivalence classes.

Let $T = (V, E)$ be a tree with $V \triangleq (0, \mathbb{R}) \cup \{(i, [x]_{\leq i}) \mid i \geq 1, x \in \mathbb{R}\}$ and

$$E \triangleq \{((0, \mathbb{R}), (1, [x]_{\leq 1})) \mid x \in \mathbb{R}\} \cup$$
$$\{((i, [x]_{\leq i}), (j, [x]_{\leq j})) \mid j > i \wedge f_j(x) \neq f_i(x) \wedge \forall i < k < j \cdot f_i(x) = f_k(x)\} .$$

That is, T is a tree of refined equivalence classes with edges corresponding to differences in f_i's. Let T_i be the subtree of T restricted to the nodes (j, X) where $j \leq i$. Then the leaves of T_i correspond to f_i, i.e., $(i, [x]_{\leq i})$ is a leaf iff $f_i(x) \neq f_{i-1}(x)$. T is finitely-branching because all edges from an equivalence class at level i only go to equivalence classes at some other level j, and there are finitely many classes at each level. Formally,

$$((i, [x]_{\leq i}), (j, [x]_{\leq j})) \in E \wedge ((i, [y]_{\leq i}), (k, [y]_{\leq k})) \in E \wedge ([x]_{\leq i} = [y]_{\leq i}) \Rightarrow j = k$$

which follows from cases 2 and 3 of (18).

Suppose $\{f_i\}_{i=1}^{\infty}$ is not stable. Then, T is infinite. By König's lemma, there is an infinite path $\pi = (0, \mathbb{R}), (1, [x]_{\leq 1}), (i_2, [x]_{\leq i_2}), \ldots$ in T. By Def. 1, for any consecutive nodes $(i_k, [x]_{\leq i_k})$ and $(i_{k+1}, [x]_{\leq i_{k+1}})$ on π, there is a $d \in D$, s.t. $f_{i_{k+1}}(x) = f_{i_k}(x) \nabla_d d$. This contradicts that ∇_d is a widening. □

Widening for BOXES. Recall that 1-dimensional BOXES are step functions into $\{\text{TRUE}, \text{FALSE}\}$. Thus, ∇_s where $\nabla_d = \vee$ is a widening for them. Widening, ∇_{bs}^n for n-dimensional BOXES is defined recursively by letting ∇_{bs}^n be ∇_s parameterized by $\nabla_d = \nabla_{bs}^{n-1}$. We write ∇_{bs} when the dimension is clear or irrelevant.

Theorem 3. *The operation ∇_{bs} is a widening for* BOXES.

We now describe our implementation of ∇_{bs} with LDDs. It is not hard to show that the last two cases of (18) are equivalent to v_{i+1} and v_{i-1}, respectively. That is, the value of the partition i is either a widening of the corresponding partitions of the arguments, or the value of an adjacent partition. Thus, if we assume that the step functions are given as a linked list of partitions, ∇_s is computable by a recursive traversal of this list. Conveniently, this is how BOXES are represented by LDDs. For example, in Fig. 1(c) the low edges form the linked list of partitions of dimension x. However, there are no back-edges, and it is hard to access the value of the "previous" partition. We overcome this problem by sending the value of the "current" partition down the recursion chain.

Our algorithm WR implementing $f \nabla_{bs}^n g$ is shown in Fig. 6. The inputs are LDDs f and g, a variable x bound to dimension n, and an LDD h representing the value of "previous" partition or **nil**. When the dimension of f and g is not known apriori, $f \nabla_{bs} g$ is implemented by WR$(f, g, x, \textbf{nil})$, where x is the $\preceq$-least variable of f and g, and h is **nil** since the algorithm starts at the first partition. This is done by WIDEN shown in Fig. 6. The WR proceeds exactly as the simple recursive algorithm described above. Comments indicate which lines correspond to the three cases of (18).

```
Require: LEQ(f, g)
 1: function WIDEN (LDD f, LDD g)
 2:     if (f = 0) ∨ (f = g) ∨ (g = 1) then
 3:     else   return g
 4:     if C(f) ⪯ C(g) then return WR(f, g, VAR(C(f)), nil)
 5:     else   return WR(f, g, VAR(C(g)), nil)
 6: function WR (LDD f, LDD g, VAR x, LDD h)
 7:     if f = g then
 8:         if (VAR(C(f)) ≠ x) ∧ (h ≠ nil) then return h                  ▷ (case 3)
 9:         return g                                                       ▷ (case 1)
10:     if (VAR(C(f)) ≠ x) ∧ (VAR(C(g)) ≠ x) then return WIDEN(f, g)      ▷ (case 1)
11:     if C(f) ⪯ C(g) then v ← C(f)
12:     else   v ← C(g)
13:     t ← WR(f|_v, g|_v, x, nil)
14:     e ← WR(f|_¬v, g|_¬v, x, nil)
15:     if v = C(f) then
16:         if (g|_v = f|_v) ∧ (h ≠ nil) then
17:             return ITE(v, h, e)                                        ▷ (case 3)
18:         else
19:             return ITE(v, t, e)                                        ▷ (case 1)
20:     if g|_v = f|_v then return e                                       ▷ (case 2)
21:     return ITE(v, t, WR(f|_¬v, g|_¬v, x, t))                           ▷ (case 1)
```

Fig. 6. Widening for BOXES

Theorem 4. *Algorithm* WIDEN *implements* ∇_{bs} *in time* $O(|f| \cdot |g|)$.

5 BOXES and Finite Powerset of BOX

The finite powerset of BOX [1,2], which we call POWERBOX, is the main alternative to BOXES as a refinement of BOX. An advantage of a finite powerset construction is its applicability to any base domain. However, this makes it hard (if not impossible) to leverage the power of domain-specific data structures. In contrast, our BOXES implementation is based on a specific data-structure – LDDs – but does not extend to other base domains. In the rest of the section, we compare the two domains analytically. Results of extensive empirical evaluation are presented in Section 6.

Finite powerset construction. Let $\hat{D} = (D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be an abstract domain. For any $S \subseteq D$, let $\Omega(S)$ be the set of the $\sqsubseteq$-maximal elements of S, and $S \subseteq_{fn} D$ mean that S is a *finite* subset of D. The *finite powerset* domain over $\hat{D}$ is:

$$\hat{D}_P = (\mathcal{P}^{\Omega}_{fn}(\hat{D}), \sqsubseteq_P, \emptyset, \Omega(D), \sqcup_P, \sqcap_P)\,, \tag{20}$$

where $\mathcal{P}^{\Omega}_{fn}(\hat{D}) \triangleq \{S \subseteq_{fn} D \mid \Omega(S) = S\}$, $S_1 \sqsubseteq_P S_2$ iff $\forall d_1 \in S_1 \cdot \exists d_2 \in S_2 \cdot d_1 \sqsubseteq d_2$, $S_1 \sqcup_P S_2 \triangleq \Omega(S_1 \cup S_2)$, and $S_1 \sqcap_P S_2 \triangleq \Omega(\{s_1 \sqcap s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\})$.

Comparing Representation. BOXES and POWERBOX differ in their element representation. Let φ be a Boolean formula over $\mathbb{IVQ}$. POWERBOX represents φ by its (unshared) DNF, while BOXES represents φ by its BDD. Thus, there exists a φ whose POWERBOX representation is exponentially bigger than its BOXES representation, and vice versa. Of course, deciding between a DNF or a BDD representation of a Boolean formula is a long-standing open problem.

Comparing Basic Operations. The $\subseteq$ operation of BOXES is exact, while the corresponding $\sqsubseteq_P$ operation of POWERBOX is not. For example, let $S_1 = \{0 \leq x < 2\}$ and $S_2 = \{0 \leq x < 1, 1 \leq x < 2\}$ be elements of POWERBOX. Then, $(S_1 \not\sqsubseteq_P S_2)$, but $S_1 \subseteq S_2$. The complexity of the operations in both domains is polynomial in the sizes of the representations of their arguments. Complexities of the LDD operations used by BOXES are shown in Table 1. For POWERBOX, most expensive operations are Ω and meet ($\sqcap_P$). Ω is quadratic and has no analogue in BOXES. $\sqcap_P$ has the same complexity, relative to the size of its arguments, as AND. The complexity of join ($\sqcup_P$) is similar to OR, but is more efficient if irreducibility of the result is not required.

Comparing Widening. Bagnara et al. [2] suggest three schemes to extend a widening from the base domain (in this case, BOX) to the finite powerset (i.e., BOXES): k-bounded, connector, and certificate-based. Our widening does not fit any of these categories. It does not bound the number of disjuncts apriori, and hence is not k-bounded. It does not compute certificates, or a box hull of its arguments, and hence is not certificate-based. It is close in spirit to connector-widening, but is not itself based on widening of a base-domain. Thus, our widening is not easily comparable to any of the suggestions of [2]. Note that extending a POWERBOX widening to BOXES is difficult. One possibility is to convert between a BOXES and a POWERBOX value, apply POWERBOX widening, and convert the value back. But, this involves an exponential blowup – number of paths in an LDD is exponential in its size. The alternative is to adapt POWERBOX widening algorithm to work directly on an LDD. This is non-trivial.

In summary, it is not obvious which of BOXES and POWERBOX is superior. In Section 6, we present empirical evidence that suggests that in practice the BOXES domain does scale better.

6 Experiments

To evaluate BOXES, we implemented a simple abstract interpreter, IRA, on top of the LLVM compiler infrastructure [14]. For every function of a given program, IRA computes invariants over all SSA variables at all loop heads using a given abstract domain. We compared four abstract domains: LDD BOXES – the domain described here; LDD BOX – BOX implemented with LDDs using BOXJOIN and the standard widening instead of OR and WIDEN, respectively; PPL BOX – BOX implemented by `Rational_Box` class of PPL [3]; and, PPL BOXES – POWERBOX implemented by `Pointset_Powerset<Rational_Box>` of PPL. For LDD-based domains, we used dynamic variable ordering.

The benchmark. We applied IRA to 25 open source programs, including `mplayer`, `CUDD`, and `make`, with over 16K functions in total. All experiments were ran on a 2.8GHz quad-core Pentium machine. Running time and memory for each function was limited to 1 minute and 512MB, respectively. Here, we report on the 5,727 functions which at least one domain required 2 or more seconds to analyse. The first two columns of Table 3(a) summarize key characteristics of the benchmark: on average there are 238 variables and 7 loop heads per function.

Table 3. (a) Benchmark summary: Vars – # of variables; Loop – # of loop heads; Invariant Sizes: DD – # of nodes in a DD, Path – # of paths, Box – # of elements in a PPL BOXES value. (b) Summary of the experimental results: %S – % Solved, T – total time, %B – % time in basic ops, %I – % time in image, %∇ – time in widen.

	Vars	Loop	DD	Path	Box
MIN	9	0	1	0	1
MAX	9,052	241	87,692	2.15E09	7,296
AVG	238	7	1,011	2.46E08	802
STDEV	492	12	3,754	5.75E08	761
MEDIAN	97	3	149	5,810	589

(a)

Domain	%S	T(m)	%B	%I	%∇
LDD BOX	99.8	4	77	23	0
PPL BOX	96.1	117	86	14	0
LDD BOXES	87.9	118	61	38	1
PPL BOXES	14.2	201	95	1	3

(b)

The last 3 columns summarize the size of the invariants computed as either LDDs, number of paths in a LDD, or number of elements in a POWERBOX value. Note that the large standard deviations indicate that the benchmark was quite heterogeneous. Overall, Table 3(a) shows that our analysis was non-trivial.

The results. Our experimental results are summarized in Table 3(b). The first two columns show the percentage of (the 5,727) functions analyzed successfully, and the time taken, respectively. The time includes only the cost of abstract domain operations, and only counts the successful cases for the corresponding domain. Each BOX domain solved over 90% of the cases. Surprisingly, LDD BOX was significantly faster. We conjecture that this is due to the large number of tracked variables in our benchmark. The size of an LDD BOX value is proportional to the number of bounded variables (dimensions), whereas that size of PPL BOX value is proportional to the, much larger, number of tracked variables.

Our LDD BOXES domain did quite well, solving close to 90% of the cases. PPL BOXES domain did not scale at all: solved under 20% and took almost double the time of LDD BOXES.

The last three columns of Table 3(b) break down the time between the basic ($\sqsubseteq, \sqcap, \sqcup$) domain operations (Basic), image computation (Image), and widening (Widen). Again, both BOX domains perform similarly, with Basic being the most expensive, while Widen is negligible. For LDD BOXES, the time is divided more evenly between Basic and Image, with a non-negligible Widen. For PPL BOXES, the time is dominated by Basic, and Widen is also significant.

Fig. 7(a) compares LDD BOX (the fastest and least precise analysis) and LDD BOXES. Clearly, additional expressivity of LDD BOXES costs additional (often, several orders of magnitude) complexity. Fig. 7(b) compares PPL BOXES and LDD BOXES (only successful cases for PPL are shown). Here, LDD BOXES is several orders of magnitude faster.

In order to understand whether the increased expressivity of LDD BOXES yields more precise results, and to evaluate the effectiveness of our widening, we measured the number of times widening points are visited during the analysis. We conjecture that a very aggressive (and, thus, imprecise) widening results in a very quick convergence and, hence, few repeated applications of widening. Fig. 7(c) compares LDD BOX and LDD BOXES. In all but 23 cases, analysis

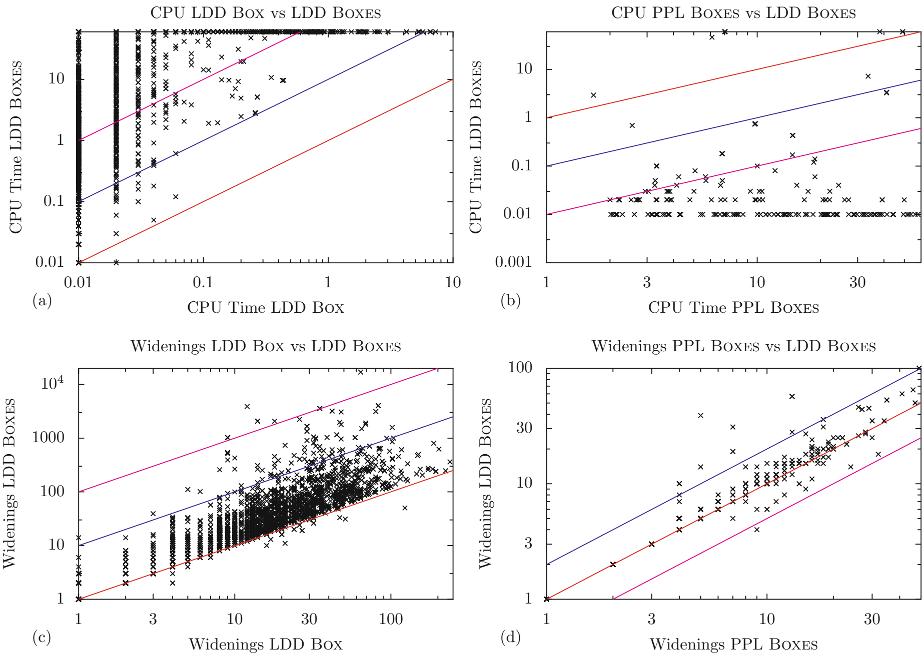


Fig. 7. Running time: (a) LDD Box vs. LDD Boxes; (b) PPL Boxes vs. LDD Boxes. Number of widenings: (c) LDD Box vs. LDD Boxes; (d) PPL Boxes vs. LDD Boxes.

with LDD Boxes visits widening points as many (and often significantly more) times than LDD Box. In the remaining 23 cases, LDD Boxes converges faster – often, before the widening is ever applied[2] – but to a more precise invariant.

Fig. 7(d) compares LDD Boxes and PPL Boxes (on the cases where PPL Boxes was successful). In most cases, both domains converge after similar number of iterations. In general, the convergence rate is within a factor of 2. We conjecture that this indicates that our widening is similar in its precision to the finite powerset widening used by PPL Boxes.

Overall, our evaluation indicates that LDDs provide a solid backbone for implementing Box and its disjunctive refinements. LDD Box is competitive with PPL Box, and scales much better as the number of variables increases. The performance degradation when moving from Box to its disjunctive refinement is milder for LDDs than for PPL. Finally, LDD Boxes performs better than PPL Boxes, while maintaining a similar precision level.

7 Conclusion

In this paper, we presented Boxes, a symbolic abstract domain that weds disjunctive refinement of Box with BDDs. Boxes is implemented on top of LDDs, an extension of BDDs to linear arithmetic. We present a novel widening algorithm for

[2] In IRA, we delay widening until the 3rd iteration of a loop.

Boxes that is different from known schemes for implementing widening for disjunctive refinements. Empirical evaluation indicates that Boxes is more scalable than existing implementations of the finite powerset of Box.

An area of future work is to study applicability and scalability of Boxes in a practical software verification setting. In particular, Boxes offers a promising platform for combining model-checking and abstract interpretation as in [12]. Another direction is to extend the approach to weakly-relational domains. The main challenge is developing an effective and efficient widening.

Acknowledgements. We thank Ofer Strichman for numerous insightful discussions.

References

1. Bagnara, R.: A Hierarchy of Constraint Systems for Data-Flow Analysis of Constraint Logic-Based Languages. Science of Computer Programming 30(1-2), 119–155 (1988)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening Operators for Powerset Domains. International Journal on Software Tools for Technology Transfer (STTT) 8(4), 449–466 (2006)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Towards A Complete Set of Numerical Abstractions for The Analysis and Verification of Hardware and Software Systems. Science of Computer Programming 72(1-2), 3–21 (2008)
4. Beyer, D., Henzienger, T.A., Theoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
5. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers (TC) 35(8), 677–691 (1986)
6. Chaki, S., Gurfinkel, A., Strichman, O.: Decision Diagrams for Linear Arithmetic. In: FMCAD 2009 (2009)
7. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Proceedings of the 2nd Internaitional Symposium on Programming (ISOP 1976), pp. 106–130 (1976)
8. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL 1979), pp. 269–282 (1979)
9. Cousot, P., Cousot, R.: Abstract Interpretation Frameworks. Journal of Logic and Computation (JLC) 2(4), 511–547 (1992)
10. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
11. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
12. Gurfinkel, A., Chaki, S.: Combining Predicate and Numeric Abstraction for Software Model Checking. In: FMCAD 2008, pp. 127–135 (2008)
13. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock Difference Diagrams. Nord. J. Comput. 6(3), 271–298 (1999)

14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO 2004 (2004)
15. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
16. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Difference Decision Diagrams. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 111–125. Springer, Heidelberg (1999)
17. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static Analysis in Disjunctive Numerical Domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006)
18. Somenzi, F.: CU Decision Diagram Package, http://vlsi.colorado.edu/~fabio/CUDD
19. Strehl, K., Thiele, L.: Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In: ICCAD 1998, pp. 686–692 (1998)
20. Wang, F.: Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 157–171. Springer, Heidelberg (2000)

Alternation for Termination

William R. Harris[1], Akash Lal[2], Aditya V. Nori[2], and Sriram K. Rajamani[2]

[1] University of Wisconsin; Madison, WI, USA
[2] Microsoft Research India; Bangalore, India

Abstract. Proving termination of sequential programs is an important problem, both for establishing the total correctness of systems and as a component of proving more general termination and liveness properties. We present a new algorithm, TREX, that determines if a sequential program terminates on all inputs. The key characteristic of TREX is that it alternates between refining an over-approximation and an under-approximation of each loop in a sequential program. In order to prove termination, TREX maintains an over-approximation of the set of states that can be reached at the head of the loop. In order to prove non-termination, it maintains an under-approximation of the set of paths through the body of the loop. The over-approximation and under-approximation are used to refine each other iteratively, and help TREX to arrive quickly at a proof of either termination or non-termination.

TREX refines the approximations in alternation by composing three different program analyses: (1) local termination provers that can quickly handle intricate loops, but not whole programs, (2) non-termination provers that analyze one cycle through a loop, but not all paths, and (3) global safety provers that can check safety properties of large programs, but cannot check liveness properties. This structure allows TREX to be instantiated using any of the pre-existing techniques for proving termination or non-termination of individual loops.

We evaluated TREX by applying it to prove termination or find bugs for a set of real-world programs and termination analysis benchmarks. Our results demonstrate that alternation allows TREX to prove termination or produce certified termination bugs more effectively than previous techniques.

1 Introduction

Proving termination of sequential programs is an important problem, both for establishing total correctness of systems and as a component for proving other liveness properties [12]. However, proving termination efficiently for general programs remains an open problem. For an illustration of the problem, consider the example program shown in Fig. 1, and in particular the loop L2 on lines 8–16. This loop terminates on all inputs, but for an analysis to prove this, it must derive two important facts: (1) the loop has an invariant $d > 0$ and (2) under this invariant, the two paths through the loop cannot execute together infinitely often. Existing analyses can discover one or the other of the above facts, but not both.

Some analyses construct a proof of termination in the form of a lexicographic linear ranking function (LLRF) [4]. These analyses can prove termination of L2 by constructing a valid LLRF if they are given $d > 0$ as a loop invariant. However, LLRF-based

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 304–319, 2010.
© Springer-Verlag Berlin Heidelberg 2010

```
1  void f(int d) {
2    int x, y, k, z := 1;
3    ...
4  L1:
5    while (z < k) { z := 2 * z; }
6    ...
7  L2:
8    while (x > 0 && y > 0) {
9      if (*) {
10 P1:    x := x - d;
11        y := *;
12        z := z - 1;
13     } else {
14        y := y - d;
15     }
16   }
17 }
```

```
1  void main() {
2    if (*) {
3      f(1);
4    } else {
5      f(2);
6    }
7  }
```

Fig. 1. Example illustrating the effect of alternation

tools have been designed to analyze only loops with affine assignments and conditions, and are unable to handle pointers, or perform inter-procedural, whole program analysis (which is required to establish the desired invariant).

Techniques that construct a transition invariant (TI) as proofs, such as TERMINATOR [10], can handle arbitrary programs with procedures and pointers, but are hampered by the way they construct a proof of termination. To illustrate this, consider how TERMINATOR analyzes loop `L2`. TERMINATOR first attempts to prove termination of `L2` by analyzing it in isolation from the rest of the program. However, TERMINATOR fails, as it is not aware of the additional fact that whenever the loop is reached, the invariant $d > 0$ holds. It thus generates a potential counterexample that may demonstrate that the loop does not always terminate. A counterexample to termination is a "lasso", which consists of a "stem" sequence of statements that executes once followed by a "cycle" sequence of statements that may then be executed infinitely often. For the example, TERMINATOR may generate a lasso with the stem "`d := 1; z := 1`" that leads to `L2`, followed by the cycle "`assume (x > 0); assume(y > 0); x := x - d; y := *; z := z - 1`" that executes infinitely often. If TERMINATOR ignores the stem, it cannot prove that the cycle will not execute infinitely often. Thus, it uses the state of the program after executing the stem, "`d = 1, z = 1`", to construct a new cycle "`assume(d = 1); assume (z = 1); assume (x > 0); assume(y > 0); x := x - d; y := *; z := z - 1`" whose behaviors under-approximate those of the original cycle. In the under-approximation, the conditions $d = 1$ and $z = 1$ are assumed to hold at the beginning of *every* iteration of the loop (see Section 3.4 of [10] for a discussion).

In this way, TERMINATOR constructs an under-approximation of the counterexample cycle in the hope that it can at least find a proof of termination for the under-approximation. With the added assumptions at the head of the cycle, it can find multiple proofs that the under-approximation eventually terminates. One such proof establishes that the expression $z-1$ is both bounded from below by 0 and must decrease through every iteration of the cycle. TERMINATOR then attempts to validate $z - 1$ as a proof of termination of the entire loop by determining if there are any paths over which $z - 1$ is not bounded and decreasing. There are, as the value of z is not bounded over the executions

of the loop. Thus TERMINATOR will find another counterexample to $z - 1$ as a proof of termination. For instance, it may find a trace that executes loop L1 once, reaches L2 with state $d = 1, z = 2$, and executes the same cycle as the previous counterexample. Similarly to how TERMINATOR handled the last counterexample, it constructs an under-approximate cycle "`assume(d = 1); assume(z = 2); assume(x > 0); assume(y > 0); x := x - d; y := *; z := z - 1;`" and attempts to prove its termination. Similar to the last counterexample, it determines that $z - 2$ is bounded from below by 0 and decreases each time through the loop. Again, this fact does not hold for all paths through the loop, so TERMINATOR will iterate again on another counterexample. In this way, TERMINATOR will converge on a proof of termination slowly, if at all.

To address these shortcomings in existing techniques, we propose TREX, a novel approach to proving termination of whole programs. TREX addresses the shortcomings of LLRF-based techniques and TERMINATOR with an algorithm that *alternates* between refining an over and under-approximation of the program. TREX analyzes loops in the program one at a time. For each loop L, it simultaneously maintains an over-approximation as a loop invariant for L (which is a superset of the states that can be reached at the loop-head) and an under-approximation as a subset of all the paths through L.

TREX first applies a loop termination prover to try to prove that no set of paths in the under-approximation can execute together infinitely often. If the loop termination prover can prove this, then it produces a *certificate* of the proof. TREX then checks if the certificate is a valid proof that no set of paths in the entire loop may execute infinitely often. If so, then the certificate demonstrates that the loop terminates on all inputs. If not, then TREX adds to the under-approximation paths that invalidate the certificate. TREX then reanalyzes the program using the new, expanded under-approximation. This technique is similar to those employed in TERMINATOR.

If TREX fails to prove that paths in the under-approximation do not execute infinitely often, then it applies a non-termination prover to find a sufficient condition for non-termination. This sufficient condition is a precondition under which the loop will *not* terminate. TREX then queries a safety prover to search for a program input that reaches the loop and satisfies this precondition. If the safety prover finds such an input, then the input is a true counterexample to termination. If the safety prover determines that the loop precondition is unreachable, then the negation of the precondition is an invariant for the loop. TREX conjoins this predicate to its existing invariant and reanalyzes the program using the new, strengthened over-approximation. This technique is novel to TREX.

In this way, TREX composes three analyzes for three distinct problems: (1) efficient local termination provers that can analyze a loop represented as a finite sets of paths, (2) non-termination provers that analyze a single trace, and (3) safety provers that prove global safety properties of programs. This composition allows each analysis to improve the performance of the other. The composition allows TREX to apply a loop termination prover that produces a *lexicographic linear ranking functions* (LLRF) as a certificate of termination. Using LLRFs as certificates, as opposed to TIs, improves the performance of the safety prover in validating certificates. The non-termination prover allows

LLRF-based loop termination provers to reason about loops that cannot be proved terminating when analyzed in isolation. Finally, the safety prover directs the search of the non-termination prover in finding counterexamples to termination. Using this approach, TREX is able to prove termination or non-termination of programs that are outside the reach of existing techniques, including the example in Fig. 1. §2 gives an informal discussion as to how TREX handles this example.

The contributions of this paper are as follows:

1. We present TREX, a novel algorithm for proving termination of whole programs. TREX simultaneously maintains over and under-approximations of a loop to quickly find proofs of termination or non-termination. This allows it to compose several program analyses that until now were disparate: termination provers for multi-path loops, non-termination provers for cycles, and global safety provers.
2. We present an empirical evaluation of TREX. We evaluated TREX by applying it to a set of systems drivers and benchmarks for termination analysis, along with versions both that we injected with faults. The results of our evaluation demonstrate that TREX's use of alternation allows it to quickly prove that programs either always terminate or produce verified counterexamples to their termination.

The rest of this paper is organized as follows. In §2, we illustrate by example how TREX proves termination or non-termination for an example program. In §3, we review known results on which TREX builds. In §4, we give a formal presentation of the TREX algorithm. In §5, we present an empirical evaluation of TREX. In §6 we discuss related work, and in §7 we conclude.

2 Overview

We now informally present the TREX algorithm. We first describe the core algorithm for deciding if a single loop in a single-procedure program terminates under all program inputs, and then illustrate the algorithm using a set of examples. If the program contains nested loops, function calls, and pointers, the algorithm can be extended. We present such extensions in §4.2.

To analyze a loop L, TREX maintains two key pieces of information: (i) a loop invariant O of L, and (ii) U, which is a subset of the set of all paths that can execute in the body of loop L. Note that paths in U can be obtained by concatenating arbitrarily many paths through L. The overapproximation O is initialized to a weak invariant such as $\mathbf{true}$, and U is initialized to an empty set of paths. TREX analyzes each loop iteratively. In each iteration, it first attempts to find a certificate that proves that no set of paths in U can execute together infinitely often, assuming the loop invariant O.

First, suppose that TREX cannot find a proof certificate. Then TREX finds a path τ that is a concatenation of paths in U such that no proof of termination of τ exists. It then uses a *non*-termination prover [14] to derive a loop precondition φ such that if the program reaches L in a state $\sigma \in \varphi$, then it will then execute τ infinitely often. TREX calls a safety prover to determine if some initial program state σ_I can reach such a σ along an execution trace. If so, then the trace, combined with τ, is a witness that the loop does not always terminate. If a safety prover determines that no such states σ_I and

σ exist, then TREX strengthens the over-approximation of O with the knowledge that φ can never hold at the head of the loop L.

Now, suppose that TREX does find a proof certificate for the under-approximation. TREX then checks to see if the certificate is valid for all paths in L. If the certificate is not valid, then TREX finds a path τ over the body of L that invalidates the certificate, and expands U to include τ. TREX then performs another refinement step using the strengthened over-approximation or expanded under-approximation. In this way, the under-approximation U is used to find potentially non-terminating cycles, and if such cycles are unreachable, this information is used to refine the over-approximation O. Dually, if the certificate for U is not a valid certificate for all the paths through L with the over-approximation O, this information is used to expand U. We now illustrate the advantages of this approach using a set of examples.

Alternation Between Over and Under-approximations. Because TREX simultaneously maintains over and under-approximations of a loop, it can often quickly find proofs that the loop terminates, even when the proofs rely on program behavior that is not local to the loop. For example, consider loop `L2` from Fig. 1. Recall from §1 that existing termination provers may have difficulty proving termination of `L2`. A technique that relies on a fixed over-approximation may not be able to discover automatically the needed loop invariant $d > 0$, but a technique that relies solely on under-approximations may struggle to find a proof of termination for the loop, as it is misled by information gathered along a trace leading to the loop.

TREX handles this example by alternating between over and under-approximations. It first tries to prove termination of the loop with an over-approximation that the loop can be reached in any state, and is unable to find such a proof. TREX thus generates a potential counterexample to termination in the form of a cycle through the loop: `assume(x > 0 && y > 0); y := y - d`. It then applies a non-termination prover to this cycle to find a sufficient condition φ such that if execution reaches the loop in a state that satisfies φ, then the subsequent execution will not terminate. The non-termination prover determines that such a sufficient condition is the predicate $d \leq 0$. TREX then queries a safety prover to decide if the condition $d \leq 0$ at `L2` is reachable, but the safety prover determines that $d \leq 0$ is in fact unreachable. Thus TREX refines the over-approximation of the loop to record that all states reachable at line 4 are in $\neg(d \leq 0) \equiv d > 0$. TREX then applies a loop termination prover to the loop under this stronger over-approximation. Such a technique quickly proves that `L2` always terminates.

Using LLRFs as Certificates for Termination Proofs. Existing techniques for proving termination of programs produce a *transition invariant* (TI) as a certificate of proof of termination, while existing termination provers for loops produce *lexicographic linear ranking functions* (LLRF). TREX is parametrized to use either TIs or LLRFs as certificates in proving termination of whole programs. This implies that it can construct a set of LLRFs that serves as a proof of termination for a whole program. While TIs are more expressive than LLRF's in that they can be used to encode proofs of termination for more loops than LLRFs, LLRFs can often be constructed faster, and the loss of expressiveness typically does not matter in practice. We find that in practice, using LLRFs

as certificates instead of TIs results in an acceptable loss of expressiveness while allowing significant gains in performance, both in finding the certificate and in validating candidate certificates.

To gain an intuition for the advantage of using LLRFs, consider again in Fig. 1 the loop `L2`. Recall that `L2` is problematic for an analysis that constructs a TI using under-approximations. However, suppose that an analysis based on constructing TIs was given $d > 0$ as a loop invariant. The analysis could then analyze the loop in isolation and would eventually find a TI that proves termination. However, the best known approach to TI synthesis constructs proofs one at a time for single paths through potentially multiple iterations of the loop. For each path, the analysis then attempts to validate the constructed proof using an expensive safety check. However, if an LLRF-based analysis is given the loop invariant $d > 0$, and both the paths "`x := x - d; y := *; z := z - 1`", and "`y := y - d`" through the loop, it can prove termination of the loop by solving a single linear constraint system. Furthermore, the validation of resulting LLRF is considerably simpler.

```
1   int d = 1;
2   int x;
3
4   if(*) d := d - 1;
5   if(*) foo();
6   ...
7   //k such conditionals
8   //without decrements of d.
9   ...
10  if(*) foo();
11  if(*) d := d - 1;
12
13  while (x > 0) {
14    x := x - d;
15  }
```

Fig. 2. Example to illustrate detecting non-termination

Proving Non-termination. Finally, TREX can find non-terminating executions efficiently. For the program in Fig. 2, suppose that the function `foo` has p paths through its body. There are thus $O(2^k p^k)$ different lassos in the program that end with the cycle at lines 13–15. Of these, only the lassos with stems that include the decrements to d at lines 4 and 11 lead to non-termination. The current best known technique for finding termination bugs, TNT [14], searches the program for lassos in an arbitrary manner. Thus TNT may only find such a bug by enumerating the entire space of lassos.

TREX can provide TNT with a goal-directed search strategy for finding termination bugs. For the program in Fig. 2, TREX first analyzes the loop at lines 13–15, and is unable to prove termination of the loop. It next attempts to find an execution for which the loop does not terminate. However, instead of applying TNT to one of the *lassos* in the program to verify it as a *complete witness to non-termination*, TREX applies TNT to the sole *path through the loop* to derive a *sufficient condition for non-termination*. For the example, TNT determines that if the loop is reached in a state that satisfies $d < 0$, then execution of the loop will not terminate. TREX then queries a safety prover to determine if a state that satisfies $d < 0$ is reachable at the head of the loop. Suppose that the function `foo` does not modify d. Modular safety checkers such as SMASH [11] can use knowledge about the target set of states $d < 0$ to build a safety summary for `foo` which states that d is not modified by `foo`. TREX uses such a prover to quickly find a path that reaches the loop head in a state that satisfies $d < 0$. It is the path that decrements d at lines 4 and 11.

3 Preliminaries

TREX builds on existing work on proving termination and non-termination. We recall some preliminaries and definitions from previous work.

3.1 Termination Certificates

TREX is parametrized by the certificates that it uses to prove termination of individual loops. A certificate typically defines a measure μ that is bounded below by zero, (i.e. $\mu \geq 0$) and decreases on every iteration of the loop. Previous work shows how to find such measures automatically using lexicographic linear ranking functions and transition invariants. The exact details of these certificates are not important for an understanding of TREX, but for the sake of completeness, their definitions are given in [15].

3.2 Proving Non-termination

Recent work [14] addresses a dual problem to proving termination, that of proving *non*-termination of a given path through a program. Let a pair of paths $(\tau_{stem}, \tau_{cycle})$ be a *lasso*. The problem of proving non-termination is to determine if it is possible for τ_{stem} to execute once followed by infinite consecutive executions of τ_{cycle}. [14] establishes that $(\tau_{stem}, \tau_{cycle})$ is non-terminating if and only if there exists a *recurrent set* of states defined as follows:

Defn 1. *For a lasso* $(\tau_{stem}, \tau_{cycle})$*, a* recurrent set φ *is a set of states such that* (i) φ *is reachable from the beginning of the program over* τ_{stem}*; and* (ii) *For every state* $\sigma \in \varphi$*, there is a state* $\sigma' \in \varphi$ *such that* σ' *can be reached from* σ *by executing* τ_{cycle}*.*

In this work, we introduce the notion of a *partial recurrent set*, which is a relaxation of a recurrent set.

Defn 2. *A set of states* φ *is a* partial recurrent set *for a sequence of statements* τ *if it satisfies clause (ii) of Defn. 1 with* τ *in place of* τ_{cycle}*.*

One can reduce the problem of finding a recurrent set for a given lasso to solving a non-linear constraint system [14]. This is the approach implemented by TNT. The TNT technique relies on a constraint template to guide the constraint solving, and gives a heuristic for iteratively refining the template until a recurrent set is found. In practice, if a recurrent set exists, then it typically can be found with a relatively small template. TNT can be easily extended to find a partial recurrent set as well.

4 Algorithm

We now formally present the TREX algorithm, given in Fig. 3. We first describe TREX for single-procedure programs without pointers, function calls, or nested loops. We describe in §4.2 an enhancement of TREX that deals with pointers, function calls, and nested loops. TREX attempts to prove termination or non-termination of each loop in isolation. When TREX analyzes each loop L, it maintains an over-approximation O,

TREX (P)
Input: Program P
Returns: $\mathbf{Termination}$ if P terminates on all inputs,
$\mathbf{NonTermination}(\tau_{stem}, \tau_{cycle})$ if P may execute τ_{stem} once, and then execute τ_{cycle} infinitely many times.

```
 1: for each loop L in the program do
 2:    O := true // Initialize over-approximation.
 3:    U := { } // Initialize under-approximation.
 4:
 5:    loop
 6:       result := GetCertificate(O, U)
 7:       if (result = Termination(C)) then
 8:          result' := CheckValidity(C, O, L)
 9:          if (result' = Valid) then
10:             break // Analyze next program loop.
11:          else if (result' = Invalid(τ)) then
12:             U = U ∪ {τ}
13:             continue
14:          end if
15:       else if (result = Cycle(τ_cycle)) then
16:          φ = PRS(τ_cycle)
17:          if Reachable(φ) then
18:             τ_stem := SafetyTrace(φ)
19:             return NonTermination(τ_stem, τ_cycle)
20:          else
21:             O := O \ φ
22:             continue
23:          end if
24:       end if
25:    end loop
26: end for
```

Fig. 3. The TREX algorithm

which is a superset of the set of states reachable at the loop head of L, and an under-approximation U, which is a subset of the paths through the loop body. At lines 2 and 3, O is initialized to $\mathbf{true}$ (denoting all states), and U is initialized to the empty set of program paths. We use L_O to denote the loop L with each path prefixed with an assumption that O holds, and similarly for U_O.

The core of the TREX algorithm iterates through the loop in lines 5-25 of Fig. 3. Inside this loop, TREX refines the over-approximation O to smaller sets of states, adds more paths to the under-approximation U, and tries to prove either termination or non-termination of the loop L. At line 6, TREX calls *GetCertificate* to find a certificate of proof for the under-approximation U.

First, suppose that the call $GetCertificate(O, U)$ returns $\mathbf{Termination}(C)$. In this case, *GetCertificate* has found a proof C that no set of paths in U execute together

infinitely often under invariant O. In this case, TREX checks if C is a valid certificate for the entire loop L_O by calling the function $CheckValidity$ in line 8. The call $CheckValidity(C, O, L)$ returns **Valid** if the certificate C is a valid proof of termination for the loop L_O. In this case, TREX determines that L terminates, and analyzes the next loop. Otherwise, $CheckValidity$ returns $\mathbf{Invalid}(\tau)$, where $\tau \in L^+ \setminus U$ is a path such that C does not prove that a cycle of τ will not execute infinitely often. In this case, TREX adds the path τ to the under-approximation U and continues to iterate.

Now suppose that $GetCertificate$ does not find a certificate for U_O and returns $\mathbf{Cycle}(\tau_{cycle})$. Here, $\tau_{cycle} \in U^+$ is a trace formed by concatenating some sequence of paths through U. At line 16, TREX calls PRS, which computes for τ_{cycle} a partial recurrent set φ. If $\sigma_J \in \varphi$, then executing τ_{cycle} from σ_J results in a state $\sigma_F \in \varphi$. Thus if φ is reachable from a program input σ_I, then program P will not terminate on σ_I. On line 17, TREX calls a safety prover to determine if such a σ_I exists. If so, then the safety prover produces a trace τ_{stem} along with an initial state that reaches φ. TREX then presents the lasso $(\tau_{stem}, \tau_{cycle})$ as a true counterexample to termination. Otherwise, has determined that φ is unreachable. Note that although TREX derived φ using an under-approximation of the set of paths through the loop, TREX checked if φ was reachable in the original program and determined that it was not. Thus TREX refines the over-approximation O by removing from O the set of states φ. TREX then performs another iteration in search of a definite proof of or counterexample to termination.

4.1 Sub-procedures Called by TREX

```
1   //x is an input variable
2   int x;
3
4   int main() {
5     while (x > 0) {
6        if(*) foo();
7        else foo();
8     }
9   }
10
11  void foo() {
12    x--;
13  }
```

Fig. 4. Example illustrating interprocedural analysis

The TREX algorithm, as presented in Fig. 3, depends on four procedures: $Reachable$, $CheckValidity$, $GetCertificate$, and PRS. Definitions of $Reachable$ and PRS are standard. $Reachable$ answers a safety query for a program, and thus can be implemented using any static analysis tool or model checker that provides either a proof of safety or counterexample trace. TREX assumes that if $Reachable$ answers a safety query, then the answer is definite, i.e., if it returns true, then the target is indeed reachable in the program, and if it returns false, then the target cannot be reached under any input. SMASH [11] is a safety prover that satisfies these requirements and we use it in our implementation of TREX. Because reachability in programs is undecidable, $Reachable$ may not always terminate, in which case TREX does not terminate. PRS constructs a partial recurrent set for an execution trace. The implementation of such a procedure that is used in our implementation of TREX is described in [14].

Procedures $GetCertificate$, and $CheckValidity$ can be instantiated to compute and validate any certificate of a termination proof, such as TIs or LLRFs. The work in [9] gives instantiations of these procedures for TIs. If the procedures are instantiated to use TIs, then the resulting version of TREX is similar to TERMINATOR, modulo the fact

that TREX uses counterexamples to refine an over-approximation of each loop, while TERMINATOR does not attempt to maintain an over-approximation. Furthermore, TREX can be instantiated to use LLRFs to reason about programs, given suitable definitions of *GetCertificate* and *CheckValidity*. In [15], we give novel implementations of such functions.

4.2 Handling Nested Loops, Function Calls and Pointers

For TREX to reason about nested loops, function calls, and pointers, it is necessary that its sub-procedures reason about these features. The procedures *Reachable* and *CheckValidity* depend primarily on a safety prover. In the context of safety, handling nested loops and function calls is a well-studied problem, and our safety checker supports such features. However, the procedures *GetCertificate* and *PRS* must be extended from their standard definitions to handle such features. Both procedures take as input a finite set of paths. The current state-of-the-art techniques for implementing *GetCertificate* and *PRS* can only reason about paths defined over a fixed set of variables and linear updates to those variables. They cannot reason about program statements that manipulate pointers, because pointer dereferences introduce non-linear behavior. Thus to apply such techniques, an analysis must first rewrite program paths that perform pointer manipulations to a semantically equivalent form expressed purely in terms of linear updates.

TREX rewrites program paths to satisfy this condition by following a strategy used in symbolic-execution tools, and also by TERMINATOR, which is to *concretize* the values of pointers. Note that all paths added to U are produced by *CheckValidity*, which takes as input an entire program, as opposed to a single loop. Thus if *CheckValidity* determines that a certificate is not valid for an entire loop L, then it produces a counterexample in the form of a lasso $(\tau_{stem}, \tau_{cycle})$, where τ_{cycle} is a path through the loop and τ_{stem} is a path up to the loop. In the absence of pointer dereferences, function calls, or nested loops, τ_{cycle} is directly added to U. In the presence of pointer dereferences, TREX rewrites the cycle before adding it to U as follows: for an instruction `*p = *q + 5` where `p` and `q` point to scalar variables `x` and `y` respectively during the execution of τ_{stem}, TREX replaces the instruction with `x = y + 5`. This amounts to under-approximating the behavior of paths through a loop by assuming that the aliasing conditions of τ_{stem} hold in every iteration of the loop.

TREX reasons about function calls and nested loops by in-lining instructions along the path τ_{cycle} before adding the path to U. For example, suppose that we apply TREX to the program in Fig. 4. In the course of analysis, TREX expands an under-approximation of the loop in lines 5–8 by adding a path through the loop, which goes through the function `foo`. To find a certificate for a new proof of termination that includes this path, TREX applies *GetCertificate* to this path, which only looks at the instructions in the path: `assume(x > 0); x = x - 1`. *GetCertificate* produces an LLRF `x`. TREX then applies *CheckValidity*, which uses an interprocedural safety analysis to verify that `x` is indeed a ranking function for the entire loop, i.e., in all executions of the program, the value of `x` decreases on every iteration of the loop.

4.3 Limitations of TREX

If TREX terminates, then it produces a proof of termination or a valid counterexample that witnesses non-termination. However, TREX may not terminate for the following reasons: (i) the underlying safety prover or non-termination prover may not terminate; or (ii) the main loop in Fig. 3 lines 5–25 may not terminate. The main loop may not terminate because finding the termination proof or non-termination witness may require TREX to reason about program features beyond what are supported by the loop termination and non-termination provers used by TREX. Such program features include non-linear arithmetic or manipulating recursive data-structures. Proving termination in the latter case is addressed in [3]. It would be interesting to instantiate TREX with the prover presented in [3], provided that a corresponding non-termination prover could be derived.

5 Experiments

We empirically evaluated TREX over a set of experiments designed to determine if:

- TREX can prove termination and find bugs for programs explicitly designed to be difficult to analyze for termination. To this end, we applied TREX to several hand-crafted benchmarks.
- TREX can prove termination and find bugs for real-world programs. To this end, we applied TREX to several drivers for the Windows Vista operating system.

To evaluate TREX, we implemented the algorithm described in §4, instantiated with the LLRF-based termination prover described in [15] and the non-termination prover described in §3.2. We also compared TREX with the current state of the art in proving termination. The only other termination prover that we are aware of that can analyze arbitrary C programs is TERMINATOR. We did not have access to the implementation of TERMINATOR discussed in [10], so we reimplemented it using the description provided in that work. We refer to this implementation as R-TERMINATOR. To allow for a fair comparison, the implementations of both TREX and R-TERMINATOR use the same safety prover, SMASH [11]. All experiments were performed on a machine with an AMD Athlon 2.2 GHz processor and 2GB RAM.

5.1 Micro-benchmarks

We first evaluated if TREX could find difficult termination bugs in small program snippets. To do so, we first applied R-TERMINATOR and TREX to the loop shown in Fig. 5, based on the program in Fig. 1. R-TERMINATOR did not find the bug in this loop: as described in §1, it successively tries as proofs of termination ranking functions $c_i - z$ for different constants c_i. TREX found this bug within 5 seconds, requiring 1 alternation. This example thus indicates that for a non-terminating loop with variables spurious to proving termination, z in Fig. 1, the spurious variables can cause R-TERMINATOR not to find a proof of termination or non-termination.

Table 1. Results of applying TREX to Windows drivers snippets. The timeout (T/O) limit was set to 500 seconds.

Name	Num Loops	Buggy Loops	TREX #NT	TREX #TC	TREX Time (s)	R-TERMINATOR #TC	R-TERMINATOR Time (s)	TREX speedup
01	3	0	0	3	13.8	4	32.1	2.3
02	3	1	1	2	15.3	5	48.0	3.1
03	1	1	1	0	7.9	1	5.9	0.7
04	1	0	0	1	3.1	1	12.3	3.9
05	1	0	0	1	6.4	1	8.8	1.4
06	1	0	0	1	3.0	2	13.8	4.6
07	2	0	0	2	10.2	2	11.8	1.2
08	2	0	0	2	9.4	2	11.0	1.2
09	2	1	–	–	T/O	–	T/O	–
10	1	0	0	1	2.5	2	10.3	4.1

```
int x,d,z;
d=0; z=0;

while(x > 0) {
   z ++;
   x = x - d;
}
```

Fig. 5. A non-terminating loop

Next, we applied TREX and R-TERMINATOR on snippets of code extracted from real Windows Vista drivers, the same used in [2]. The results of the experiments are given in Tab. 1. For each driver snippet, Tab. 1 reports the number of loops, the number of buggy (non-terminating) loops, the number of times that TREX called a non-termination prover during analysis (#NT), the number of times TREX called a termination prover (#TC), the time taken by TREX, and similarly for R-TERMINATOR. In general, TREX was significantly faster than R-TERMINATOR. In most cases, the speedup was caused directly by the fact that TREX uses LLRF's as termination certificates, whereas R-TERMINATOR uses TI's. By using LLRF's, TREX needs to construct fewer certificates during analysis, and thus needs to query a safety prover fewer times in order to validate certificates.

For these programs, TREX called its non-termination prover at most once. In each case, the call verified that the loop is indeed non-terminating. Program "02" highlights the advantage of applying a non-termination prover in this way. When analyzing program "02," R-TERMINATOR constructed and failed to validate multiple candidate termination certificates obtained by under-approximating the behavior of cycles. R-TERMINATOR eventually could not construct a new candidate and reported a possible termination bug. When applied to program "02," TREX failed to find a proof of termination, but then immediately alternated to apply a non-termination prover, which quickly found a verified termination bug. Finally, note that program "09" has a complicated loop about which neither TREX nor R-TERMINATOR can find a proof, and thus time out.

The original driver snippets contain relatively few termination bugs. Thus to further measure TREX's ability to find bugs, we modified each driver snippet that had no termination bug as follows. We introduced variables "inc_1, inc_2, ...", and code that non-deterministically initializes them to 0 or 1. We then replaced increment or decrement statements of the form "x = x $\pm$ 1", with "x = x $\pm$ inc_n", where a different "n" is used for each increment statement. The results are given in Table 2. Note that our modification did not always introduce a termination bug, as in some cases, the increment or decrement was irrelevant to the termination argument for the loop.

In general, TREX and R-TERMINATOR analyze these loops in similar amounts of time. In cases where TREX completed in less time than R-TERMINATOR, it was typically because R-TERMINATOR produced and then failed to validate more candidate

Table 2. Results of experiments over driver snippets modified to contain termination bugs

Name	Num Loops	Buggy Loops	TREX # NT	TREX # TCs	TREX Time (s)	R-TERMINATOR # TCs	R-TERMINATOR Time (s)
01	3	0	0	3	22.3	3	19.9
04	1	1	1	0	4.9	1	5.4
05	1	1	1	0	7.1	1	9.1
06	1	1	1	1	9.7	2	12.1
07	2	0	0	2	7.6	2	9.8
08	2	1	1	1	8.1	1	7.4
10	1	1	1	0	9.8	0	4.4

termination certificates. In such cases, R-TERMINATOR would typically choose as a ranking function a variable "x", where a statement such as "x = x - 1" had been modified to "x = x - inc" and "inc" was initialized to 1 on some but not all paths through the loop. R-TERMINATOR would only discover later in its analysis, after an expensive safety query, that "x" need not always decrease. In contrast, TREX did not choose "x" as a ranking function in this case because it never considers the concrete values of the "inc" variables while trying to find a ranking function. We believe that the difference in performance between TREX and R-TERMINATOR would increase for when applied to larger programs containing bugs as described above. This is because it typically takes less time to answer a non-termination query than it does safety query, as the former is a local property of a loop while the latter is a global property of a program.

```
int x1,x2, ..., xn;
int d1,d2, ..., dn;
d1 = d2 = ... = dn = 1;

while(x1 > 0 && x2 > 0
      && ... && xn > 0) {
  if(*) x1 = x1 - d1;
  else if(*) x2 = x2 - d2;
  ...
  else xn = xn - dn;
}
```

(a)

n	TREX Time (s)	#NT	#TC	Num. Alts.
1	9.9	1	1	2
2	11.9	2	2	4
3	27.7	3	3	6
4	97.4	4	4	8
5	396.6	5	5	10

(b)

Fig. 6. (a) A family of loops requiring significant alternation to analyze. (b) TREX results.

A Micro-benchmark Forcing Alternation. We evaluated the performance of TREX when analyzing loops for which multiple alternations are required to find a proof of termination or a bug. Consider the class of loops defined in Fig. 6. Each value of n defines a loop. To prove such a loop terminating, TREX must perform $2n$ alternations between searching for an LLRF to prove the loop terminating and searching for a PRS to prove the loop non-terminating. The results of applying TREX to the loops defined by $n \in [1, 5]$ are given in Fig. 6(b). TREX found a proof of termination in each case. The results indicate that alternation between the LLRF search and PRS search scales quite well for up to 6 alternations, but that performance begins to degrade rapidly when the analysis requires more than 8 alternations. In practice, this is not an issue, as most loops require less than 3 alternations to analyze.

We also applied R-TERMINATOR to these programs, but R-TERMINATOR timed out in each case. In its analysis, R-TERMINATOR under-approximates cycles in order to produce the x_i as candidates for proofs of termination. However, when R-TERMINATOR applies a safety prover to validate these candidates, the safety prover does not terminate. This is because the safety prover, based on predicate abstraction, uses a weakest precondition operator to find predicates relevant to its analysis. In the safety queries made by R-TERMINATOR, these predicates are not sufficient: the safety prover needs an additional predicate $d_i > 0$ to establish that some x_i decreases each time through the loop. In contrast, TREX uses a non-termination prover to find that $d_i \leq 0$, and thus establishes that $d_i > 0$ as a loop invariant. Thus when TREX makes a subsequent call to the safety prover, the call terminates.

We evaluated TREX's ability to find bugs for such a loop. For the loop defined by $n = 5$, we injected a fault that initialized $d_3 = 0$. For this loop, TREX found the resulting termination bug using 5 alternations in 22.2 seconds.

Table 3. Results of experiments over Windows Drivers. Time out was set to 1 hour.

Name	LOC	#Loops	TREX Time (s)	R-TERMINATOR Time (s)
Driver-1	0.8K	2	80	85
Driver-2	2.3K	4	1128	2400
Driver-3	3.0K	10	54	120
Driver-4	5.3K	17	945	T/O
Driver-5	6.0K	24	24	T/O
Driver-6	6.5K	16	68	62

5.2 Windows Drivers

We applied TREX to complete Windows Drivers to evaluate its ability to analyze programs of moderate size that manipulate pointers and contain multiple procedures. The drivers were chosen randomly from the Microsoft's Static Driver Verifier regression suite. We could not directly compare TREX to R-TERMINATOR over the drivers used in [10], as these were not available. The results of the evaluation are given in Table 3. The drivers used are well-tested, and thus we did not find any bugs in them. However, the results show that TREX is faster than R-TERMINATOR in most cases. Similar to the micro-benchmarks presented in §5.1, this is because R-TERMINATOR produced many more termination certificates, resulting in more safety queries.

6 Related Work

TREX brings together threads of work in proving termination that were disparate up to now. Our work shares the most in common with TERMINATOR [10]. TERMINATOR iteratively reasons about under-approximations of a program to construct a proof of termination. TREX simultaneously refines under and over-approximations of a program.

TREX relies on an analysis that proves termination of loops represented as a set of guarded linear transformations of program variables. Many existing techniques prove termination of such loops by constructing linear ranking functions [2,4,5,6,7,16]. Such techniques are efficient, but can only be applied to a restricted class of loops and cannot

reason about the contexts in which loops execute. In this work, we show how all such techniques can be brought be bear in analyzing general programs, provided they can be extended to generate counterexample traces on failure. In [15], we describe how to do this based on the technique of [4].

The technique of [4], while constructing a termination proof, uses constraint solving to find a loop invariant that is used to prove termination. It would be interesting to see how this can be used inside TREX that additionally uses a safety prover to generate invariants. We leave this as future work.

TREX also relies on techniques that prove that a given lasso does not terminate [14]. TREX applies such a technique to simultaneously search for counterexamples to termination and to guide the search for a proof of termination. TREX can also be used as a search strategy for finding non-termination bugs. The search strategy proposed in [14] simply enumerates potential cycles using symbolic execution. [8] gives a method for deriving a sufficient precondition for a loop to terminate. However, this approach does not lend well to refinement if the computed precondition is not met. TREX applies [14] iteratively to derive a sufficient precondition for termination that is guaranteed to be a true precondition of a loop.

Multiple safety provers [1,11,13] demonstrate that alternating between over and under approximations is more effective for proving safety properties than an analysis based exclusively on one or the other. For these provers, an over-approximation of the program is an abstraction of its transition relation, and the under-approximation is a set of tests through the program. The abstraction directs test generation, while the tests guide which parts of the abstraction are refined. TREX demonstrates that the insight of maintaining over and under approximations can be applied to prove termination properties of programs as well. However, for TREX, the over-approximation maintained is an invariant for a loop under analysis, and the under-approximation is a set of concrete paths through the loop. The invariant directs what new paths through the loop are considered, and the concrete paths guide the refinement of the loop invariant.

7 Conclusion

Safety provers that simultaneously refine under and over-approximations of a program can often prove safety properties of programs effectively. In this work, we have shown that the same refinement scheme can be applied to prove termination properties of programs. We derived an analysis based on this principle, implemented it, and applied it to a set of termination analysis benchmarks and real-world systems code. Our results demonstrate that alternation between approximations significantly improves the effectiveness and performance of termination analysis.

References

1. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: ISSTA, pp. 3–14 (2008)
2. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. In: POPL, pp. 211–224. ACM, New York (2007)

3. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
5. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination analysis of integer linear loops. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 488–502. Springer, Heidelberg (2005)
7. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking abstractions. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 148–162. Springer, Heidelberg (2008)
8. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
9. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426 (2006)
11. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional must program analysis: Unleashing the power of alternation. In: POPL, pp. 43–56 (2010)
12. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: POPL, pp. 16–28 (2009)
13. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: FSE, pp. 117–127 (2006)
14. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: POPL, pp. 147–158 (2008)
15. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for Termination. Technical Report MSR-TR-2010-61, Microsoft Research India (May 2010)
16. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004)

Interprocedural Analysis with Lazy Propagation

Simon Holm Jensen[1,⋆], Anders Møller[1,⋆,⋆⋆], and Peter Thiemann[2]

[1] Aarhus University, Denmark
{simonhj,amoeller}@cs.au.dk
[2] Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract. We propose *lazy propagation* as a technique for flow- and context-sensitive interprocedural analysis of programs with objects and first-class functions where transfer functions may not be distributive. The technique is described formally as a systematic modification of a variant of the monotone framework and its theoretical properties are shown. It is implemented in a type analysis tool for JavaScript where it results in a significant improvement in performance.

1 Introduction

With the increasing use of object-oriented scripting languages, such as JavaScript, program analysis techniques are being developed as an aid to the programmers [7, 8, 29, 27, 2, 9]. Although programs written in such languages are often relatively small compared to typical programs in other languages, their highly dynamic nature poses difficulties to static analysis. In particular, JavaScript programs involve complex interplays between first-class functions, objects with modifiable prototype chains, and implicit type coercions that all must be carefully modeled to ensure sufficient precision.

While developing a program analysis for JavaScript [14] aiming to statically infer type information we encountered the following challenge: *How can we obtain a flow- and context-sensitive interprocedural dataflow analysis that accounts for mutable heap structures, supports objects and first-class functions, is amenable to non-distributive transfer functions, and is efficient and precise?* Various directions can be considered. First, one may attempt to apply the classical monotone framework [18] as a whole-program analysis with an iterative fixpoint algorithm, where function call and return flow is treated as any other dataflow. This approach turns out to be unacceptable: the fixpoint algorithm requires too many iterations, and precision may suffer because spurious dataflow appears via interprocedurally unrealizable paths. Another approach is to apply the IFDS technique [23], which eliminates those problems. However, it is restricted to distributive analyses, which makes it inapplicable in our situation. A further

⋆ Supported by The Danish Research Council for Technology and Production, grant no. 274-07-0488.

⋆⋆ Corresponding author.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 320–339, 2010.
© Springer-Verlag Berlin Heidelberg 2010

consideration is the functional approach [26] which models each function in the program as a partial summary function that maps input dataflow facts to output dataflow facts and then uses this summary function whenever the function is called. However, with a dataflow lattice as large as in our case it becomes difficult to avoid reanalyzing each function a large number of times. Although there are numerous alternatives and variations of these approaches, we have been unable to find one in the literature that adequately addresses the challenge described above. Much effort has also been put into more specialized analyses, such as pointer analysis [10], however it is far from obvious how to generalize that work to our setting.

As an introductory example, consider this fragment of a JavaScript program:

```
function Person(n) { this.setName(n); }
Person.prototype.setName = function(n) { this.name = n; }
function Student(n,s) { Person.call(this, n);
                        this.studentid = s.toString(); }
Student.prototype = new Person;
var x = new Student("John Doe", 12345);
x.setName("John Q. Doe");
```

The code defines two "classes" with constructors `Person` and `Student`. `Person` has a method `setName` via its prototype object, and `Student` inherits `setName` and defines an additional field `studentid`. The `call` statement in `Student` invokes the super class constructor `Person`.

Analyzing the often intricate flow of control and data in such programs requires detailed modeling of points-to relations among objects and functions and of type coercion rules. TAJS is a whole-program analysis based on the monotone framework that follows this approach, and our first implementation is capable of analyzing complex properties of many JavaScript programs. However, our experiments have shown a considerable redundancy of computation during the analysis that causes simple functions to be analyzed a large number of times. If, for example, the `setName` method is called from other locations in the program, then the slightest change of any abstract state appearing at any call site of `setName` during the analysis would cause the method to be reanalyzed, even though the changes may be entirely irrelevant for that method. In this paper, we propose a technique for avoiding much of this redundancy while preserving, or even improving, the precision of the analysis. Although our main application is type analysis for JavaScript, we believe the technique is more generally applicable to analyses for object-oriented languages.

The main idea is to introduce a notion of "unknown" values for object fields that are not accessed within the current function. This prevents much irrelevant information from being propagated during the fixpoint computation. The analysis initially assumes that no fields are accessed when flow enters a function. When such an unknown value is read, a recovery operation is invoked to go back through the call graph and propagate the correct value. By avoiding to recover the same values repeatedly, the total amortized cost of recovery is never higher

than that of the original analysis. With large abstract states, the mechanism makes a noticeable difference to the analysis performance.

Lazy propagation should not be confused with demand-driven analysis [13]. The goal of the latter is to compute the results of an analysis only at specific program points thereby avoiding the effort to compute a global result. In contrast, lazy propagation computes a model of the state for each program point.

The contributions of this paper can be summarized as follows:

- We propose an ADT-based adaptation of the monotone framework to programming languages with mutable heap structures and first-class functions and exhibit some of its limitations regarding precision and performance.
- We describe a systematic modification of the framework that introduces *lazy propagation*. This novel technique propagates dataflow facts "by need" in an iterative fixpoint algorithm. We provide a formal description of the method to reason about its properties and to serve as a blueprint for an implementation.
- The lazy propagation technique is experimentally validated: It has been implemented into our type analysis for JavaScript, TAJS [14], resulting in a significant improvement in performance.

In the full version of the paper [15], we also prove termination, relate lazy propagation with the basic framework—showing that precision does not decrease, and sketch a soundness proof of the analysis.

2 A Basic Analysis Framework

Our starting point is the classical monotone framework [18] tailored to programming languages with mutable heap structures and first-class functions. The mutable state consists of a heap of objects. Each object is a map from field names to values, and each value is either a reference to an object, a function, or some primitive value. Note that this section contains no new results, but it sets the stage for presenting our approach in Section 3.

2.1 Analysis Instances

Given a program Q, an instance of the monotone framework for an analysis of Q is a tuple $\mathcal{A} = (F, N, L, P, C, n_0, c_0, \mathsf{Base}, T)$ consisting of:

F: the set of *functions* in Q;
N: the set of *primitive statements* (also called *nodes*) in Q;
L: a set of *object labels* in Q;
P: a set of *field names* (also called *properties*) in Q;
C: a set of abstract *contexts*, which are used for context sensitivity;
$n_0 \in N$ and $c_0 \in C$: an initial statement and context describing the entry of Q;
Base: a *base lattice* for modeling primitive values, such as integers or booleans;

$T : C \times N \to \mathsf{AnalysisLattice} \to \mathsf{AnalysisLattice}$: a monotone *transfer function* for each primitive statement, where $\mathsf{AnalysisLattice}$ is a lattice derived from the above information as detailed in Section 2.2.

Each of the sets must be finite and the Base lattice must have finite height. The primitive statements are organized into intraprocedural control flow graphs [19], and the set of object labels is typically determined by allocation-site abstraction [16,5].

The notation $\mathit{fun}(n) \in F$ denotes the function that contains the statement $n \in N$, and $\mathit{entry}(f)$ and $\mathit{exit}(f)$ denote the unique entry statement and exit statement, respectively, of the function $f \in F$. For a function call statement $n \in N$, $\mathit{after}(n)$ denotes the statement being returned to after the call. A *location* is a pair (c, n) of a context $c \in C$ and a statement $n \in N$.

2.2 Derived Lattices

An analysis instance gives rise to a collection of derived lattices. In the following, each function space is ordered pointwise and each powerset is ordered by inclusion. For a lattice X, the symbols $\bot_X$, $\sqsubseteq_X$, and $\sqcup_X$ denote the bottom element (representing the absence of information), the partial order, and the least upper bound operator (for merging information). We omit the X subscript when it is clear from the context.

An *abstract value* is described by the lattice Value as a set of object labels, a set of functions, and an element from the base lattice:

$$\mathsf{Value} = \mathcal{P}(L) \times \mathcal{P}(F) \times \mathsf{Base}$$

An *abstract object* is a map from field names to abstract values:

$$\mathsf{Obj} = P \to \mathsf{Value}$$

An *abstract state* is a map from object labels to abstract objects:

$$\mathsf{State} = L \to \mathsf{Obj}$$

Call graphs are described by this powerset lattice:

$$\mathsf{CallGraph} = \mathcal{P}(C \times N \times C \times F)$$

In a call graph $g \in \mathsf{CallGraph}$, we interpret $(c_1, n_1, c_2, f_2) \in g$ as a potential function call from statement n_1 in context c_1 to function f_2 in context c_2.

Finally, an element of $\mathsf{AnalysisLattice}$ provides an abstract state for each context and primitive statement (in a forward analysis, the program point immediately *before* the statement), combined with a call graph:

$$\mathsf{AnalysisLattice} = (C \times N \to \mathsf{State}) \times \mathsf{CallGraph}$$

In practice, an analysis may involve additional lattice components such as an abstract stack or extra information associated with each abstract object or field. We omit such components to simplify the presentation as they are irrelevant to the features that we focus on here. Our previous paper [14] describes the full lattices used in our type analysis for JavaScript.

$solve(\mathcal{A})$ **where** $\mathcal{A} = (F, N, L, P, C, n_0, c_0, \mathsf{Base}, T)$:
  $a := \bot_{\mathsf{AnalysisLattice}}$
  $W := \{(c_0, n_0)\}$
  while $W \neq \emptyset$ **do**
    select and remove (c, n) from W
    $T_a(c, n)$
  end while
  return a

Fig. 1. The worklist algorithm. The worklist contains *locations*, i.e., pairs of a context and a statement. The operation $T_a(c, n)$ computes the transfer function for (c, n) on the current analysis lattice element a and updates a accordingly. Additionally, it may add new entries to the worklist W. The transfer function for the initial location (c_0, n_0) is responsible for creating the initial abstract state.

2.3 Computing the Solution

The *solution* to $\mathcal{A}$ is the least element $a \in \mathsf{AnalysisLattice}$ that solves these constraints:

$$\forall c \in C, n \in N : T(c, n)(a) \sqsubseteq a$$

Computing the solution to the constraints involves fixpoint iteration of the transfer functions, which is typically implemented with a worklist algorithm as the one presented in Figure 1. The algorithm maintains a worklist $W \subseteq C \times N$ of locations where the abstract state has changed and thus the transfer function should be applied. Lattice elements representing functions, in particular $a \in \mathsf{AnalysisLattice}$, are generally considered as *mutable* and we use the notation $T_a(c, n)$ for the assignment $a := T(c, n)(a)$. As a side effect, the call to $T_a(c, n)$ is responsible for adding entries to the worklist W, as explained in Section 2.4. This slightly unconventional approach to describing fixpoint iteration simplifies the presentation in the subsequent sections.

Note that the solution consists of both the computed call graph and an abstract state for each location. We do not construct the call graph in a preliminary phase because the presence of first-class functions implies that dataflow facts and call graph information are mutually dependent (as evident from the example program in Section 1).

This fixpoint algorithm leaves two implementation choices: the order in which entries are removed from the worklist W, which can greatly affect the number of iterations needed to reach the fixpoint, and the representation of lattice elements, which can affect both time and memory usage. These choices are, however, not the focus of the present paper (see, e.g. [17,19,12,3,28]).

2.4 An Abstract Data Type for Transfer Functions

To precisely explain our modifications of the framework in the subsequent sections, we treat AnalysisLattice as an imperative ADT (abstract data type) [20] with the following operations:

– $getfield : C \times N \times L \times P \rightarrow \mathsf{Value}$
– $getcallgraph : () \rightarrow \mathsf{CallGraph}$
– $getstate : C \times N \rightarrow \mathsf{State}$
– $propagate : C \times N \times \mathsf{State} \rightarrow ()$
– $funentry : C \times N \times C \times F \times \mathsf{State} \rightarrow ()$
– $funexit : C \times N \times C \times F \times \mathsf{State} \rightarrow ()$

We let $a \in \mathsf{AnalysisLattice}$ denote the current, mutable analysis lattice element. The transfer functions can only access a through these operations.

The operation $getfield(c, n, \ell, p)$ returns the abstract value of the field p in the abstract object ℓ at the entry of the primitive statement n in context c. In the basic framework, *getfield* performs a simple lookup, without any side effects on the analysis lattice element:

$a.getfield(c \in C, n \in N, \ell \in L, p \in P)$:
  return $u(\ell)(p)$ **where** $(m, _) = a$ **and** $u = m(c, n)$

The *getcallgraph* operation selects the call graph component of the analysis lattice element:

$a.getcallgraph()$:
  return g **where** $(_, g) = a$

Transfer functions typically use the *getcallgraph* operation in combination with the *funexit* operation explained below. Moreover, the *getcallgraph* operation plays a role in the extended framework presented in Section 3.

The *getstate* operation returns the abstract state at a given location:

$a.getstate(c \in C,\ n \in N)$:
  return $m(c, n)$ **where** $(m, _) = a$

The transfer functions must not read the field values from the returned abstract state (for that, the *getfield* operation is to be used). They may construct parameters to the operations *propagate*, *funentry*, and *funexit* by updating a copy of the returned abstract state.

The transfer functions must use the operation $propagate(c, n, s)$ to pass information from one location to another within the same function (excluding recursive function calls). As a side effect, *propagate* adds the location (c, n) to the worklist W if its abstract state has changed. In the basic framework, *propagate* is defined as follows:

$a.propagate(c \in C,\ n \in N,\ s \in \mathsf{State})$:
  let $(m, g) = a$
  if $s \not\sqsubseteq m(c, n)$ **then**
    $m(c, n) := m(c, n) \sqcup s$
    $W := W \cup \{(c, n)\}$
  end if

The operation $funentry(c_1, n_1, c_2, f_2, s)$ models function calls in a forward analysis. It modifies the analysis lattice element a to reflect the possibility of a function

call from a statement n_1 in context c_1 to a function entry statement $entry(f_2)$ in context c_2 where s is the abstract state after parameter passing. (With languages where parameters are passed via the stack, which we ignore here, the lattice is augmented accordingly.) In the basic framework, *funentry* adds the call edge from (c_1, n_1) to (c_2, f_2) and propagates s into the abstract state at the function entry statement $entry(f_2)$ in context c_2:

$a.funentry(c_1 \in C,\, n_1 \in N,\, c_2 \in C,\, f_2 \in F,\, s \in \mathsf{State})$:
 $g := g \cup \{(c_1, n_1, c_2, f_2)\}$ **where** $(_, g) = a$
 $a.propagate(c_2, entry(f_2), s)$
 $a.funexit(c_1, n_1, c_2, f_2, m(c_2, exit(f_2)))$

Adding a new call edge also triggers a call to *funexit* to establish dataflow from the function exit to the successor of the new call site.

The operation $funexit(c_1, n_1, c_2, f_2, s)$ is used for modeling function returns. It modifies the analysis lattice element to reflect the dataflow of s from the exit of a function f_2 in callee context c_2 to the successor of the call statement n_1 with caller context c_1. The basic framework does so by propagating s into the abstract state at the latter location:

$a.funexit(c_1 \in C,\, n_1 \in N,\, c_2 \in C,\, f_2 \in F,\, s \in \mathsf{State})$:
 $a.propagate(c_1, after(n_1), s)$

The parameters c_2 and f_2 are not used in the basic framework; they will be used in Section 3. The transfer functions obtain the connections between callers and callees via the *getcallgraph* operation explained earlier. If using an augmented lattice where the call stack is also modeled, that component would naturally be handled differently by *funexit* simply by copying it from the call location (c_1, n_1), essentially as local variables are treated in, for example, IFDS [23].

This basic framework is sufficiently general as a foundation for many analyses for object-oriented programming languages, such as Java or C#, as well as for object-based scripting languages like JavaScript as explained in Section 4. At the same time, it is sufficiently simple to allow us to precisely demonstrate the problems we attack and our solution in the following sections.

2.5 Problems with the Basic Analysis Framework

The first implementation of TAJS, our program analysis for JavaScript, is based on the basic analysis framework. Our initial experiments showed, perhaps not surprisingly, that many simple functions in our benchmark programs were analyzed over and over again (even for the same calling contexts) until the fixpoint was reached.

For example, a function in the `richards.js` benchmark from the V8 collection was analyzed 18 times when new dataflow appeared at the function entry:

```
TaskControlBlock.prototype.markAsRunnable = function () {
  this.state = this.state | STATE_RUNNABLE;
};
```

Most of the time, the new dataflow had nothing to do with the `this` object or the `STATE_RUNNABLE` variable. Although this particular function body is very short, it still takes time and space to analyze it and similar situations were observed for more complex functions and in other benchmark programs.

In addition to this abundant redundancy, we observed – again not surprisingly – a significant amount of spurious dataflow resulting from interprocedurally invalid paths. For example, if the function above is called from two different locations, with the same calling context, their entire heap structures (that is, the State component in the lattice) become joined, thereby losing precision.

Another issue we noticed was time and space required for propagating the initial state, which consists of 161 objects in the case of JavaScript. These objects are mutable and the analysis must account for changes made to them by the program. Since the analysis is both flow- and context-sensitive, a typical element of AnalysisLattice carries a lot of information even for small programs.

Our first version of TAJS applied two techniques to address these issues: (1) Lattice elements were represented in memory using *copy-on-write* to make their constituents shared between different locations until modified. (2) The lattice was extended to incorporate a simple effect analysis called *maybe-modified*: For each object field, the analysis would keep track of whether the field might have been modified since entering the current function. At function exit, field values that were definitely not modified by the function would be replaced by the value from the call site. As a consequence, the flow of unmodified fields was not affected by function calls. Although these two techniques are quite effective, the lazy propagation approach that we introduce in the next section supersedes the maybe-modified technique and renders copy-on-write essentially superfluous. In Section 4 we experimentally compare lazy propagation with both the basic framework and the basic framework extended with the copy-on-write and maybe-modified techniques.

3 Extending the Framework with Lazy Propagation

To remedy the shortcomings of the basic framework, we propose an extension that can help reducing the observed redundancy and the amount of information being propagated by the transfer functions. The key idea is to ensure that the fixpoint solver *propagates information "by need"*. The extension consists of a systematic modification of the ADT representing the analysis lattice. This modification implicitly changes the behavior of the transfer functions without touching their implementation.

3.1 Modifications of the Analysis Lattice

In short, we modify the analysis lattice as follows:

1. We introduce an additional abstract value, unknown. Intuitively, a field p of an object has this value in an abstract state associated with some location in

a function f if the value of p is not known to be needed (that is, referenced) in f or in a function called from f.
2. Each call edge is augmented with an abstract state that captures the data flow along the edge after parameter passing, such that this information is readily available when resolving unknown field values.
3. A special abstract state, none, is added, for describing absent call edges and locations that may be unreachable from the program entry.

More formally, we modify three of the sub-lattices as follows:

$$\mathsf{Obj} = P \to \left(\mathsf{Value}\!\downarrow_{\mathsf{unknown}}\right)$$

$$\mathsf{CallGraph} = C \times N \times C \times F \to (\mathsf{State}\!\downarrow_{\mathsf{none}})$$

$$\mathsf{AnalysisLattice} = \left(C \times N \ \to \ (\mathsf{State}\!\downarrow_{\mathsf{none}})\right) \times \mathsf{CallGraph}$$

Here, $X\!\downarrow_{\mathsf{y}}$ means the lattice X lifted over a new bottom element y. In a call graph $g \in \mathsf{CallGraph}$ in the original lattice, the presence of an edge $(c_1, n_1, c_2, f_2) \in g$ is modeled by $g'(c_1, n_1, c_2, f_2) \neq \mathsf{none}$ for the corresponding call graph g' in the modified lattice. Notice that $\bot_{\mathsf{State}}$ is now the function that maps all object labels and field names to unknown, which is different from the element none.

3.2 Modifications of the Abstract Data Type Operations

Before we describe the systematic modifications of the ADT operations we motivate the need for an auxiliary operation, *recover*, on the ADT:

$$\mathit{recover} : C \times N \times L \times P \to \mathsf{Value}$$

Suppose that, during the fixpoint iteration, a transfer function $T_a(c, n)$ invokes $a.\mathit{getfield}(c, n, \ell, p)$ with the result unknown. This result indicates the situation that the field p of an abstract object ℓ is referenced at the location (c, n), but the field value has not yet been propagated to this location due to the lazy propagation. The *recover* operation can then compute the proper field value by performing a specialized fixpoint computation to propagate just that field value to (c, n). We explain in Section 3.3 how *recover* is defined.

The *getfield* operation is modified such that it invokes *recover* if the desired field value is unknown, as shown in Figure 2. The modification may break monotonicity of the transfer functions, however, the analysis still produces the correct result [15].

Similarly, the *propagate* operation needs to be modified to account for the lattice element none and for the situation where unknown is joined with an ordinary element. The latter is accomplished by using *recover* whenever this situation occurs. The resulting operation *propagate'* is shown in Figure 3.

We then modify $\mathit{funentry}(c_1, n_1, c_2, f_2, s)$ such that the abstract state s is propagated "lazily" into the abstract state at the primitive statement $\mathit{entry}(f_2)$ in context c_2. Here, laziness means that every field value that, according to a, is not referenced within the function f_2 in context c_2 gets replaced by unknown in the abstract state. Additionally, the modified operation records the abstract

```
a.getfield′(c ∈ C, n ∈ N, ℓ ∈ L, p ∈ P):
  if m(c, n) ≠ none where (m, _) = a then
    v := a.getfield(c, n, ℓ, p)
    if v = unknown then
      v := a.recover(c, n, ℓ, p)
    end if
    return v
  else
    return ⊥Value
  end if
```

Fig. 2. Algorithm for *getfield*$'(c, n, \ell, p)$. This modified version of *getfield* invokes *recover* in case the desired field value is unknown. If the state is none according to a, the operation simply returns $\bot_{\mathsf{Value}}$.

```
a.propagate′(c ∈ C, n ∈ N, s ∈ State):
  let (m, g) = a and u = m(c, n)
  s′ := s
  if u ≠ none then
    for all ℓ ∈ L, p ∈ P do
      if u(ℓ)(p) = unknown ∧ s(ℓ)(p) ≠ unknown then
        u(ℓ)(p) := a.recover(c, n, ℓ, p)
      else if u(ℓ)(p) ≠ unknown ∧ s(ℓ)(p) = unknown then
        s′(ℓ)(p) := a.recover(c, n, ℓ, p)
      end if
    end for
  end if
  a.propagate(c, n, s′)
```

Fig. 3. Algorithm for *propagate*$'(c, n, s)$. This modified version of *propagate* takes into account that field values may be unknown in both a and s. Specifically, it uses *recover* to ensure that the invocation of *propagate* in the last line never computes the least upper bound of unknown and an ordinary field value. The treatment of unknown values in s assumes that s is recoverable with respect to the current location (c, n). If the abstract state at (c, n) is none (the least element), then that gets updated to s.

state at the call edge as required in the modified CallGraph lattice. The resulting operation *funentry*$'$ is defined in Figure 4. (Without loss of generality, we assume that the statement at $exit(f_2)$ returns to the caller without modifying the state.) As consequence of the modification, unknown field values get introduced into the abstract states at function entries.

The *funexit* operation is modified such that every unknown field value appearing in the abstract state being returned is replaced by the corresponding field value from the call edge, as shown in Figure 5. In JavaScript, entering a function body at a functions call affects the heap, which is the reason for using the state from the call edge rather than the state from the call statement. If we extended the lattice to also model the call stack, then that component would naturally be recovered from the call statement rather than the call edge.

$a.funentry'(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  let $(m, g) = a$ **and** $u = m(c_2, entry(f_2))$
  // update the call edge
  $g(c_1, n_1, c_2, f_2) := g(c_1, n_1, c_2, f_2) \sqcup s$
  // introduce unknown *field values*
  $s' := \bot_{\mathsf{State}}$
  if $u \neq$ none **then**
    for all $\ell \in L, p \in P$ **do**
      if $u(\ell)(p) \neq$ unknown **then**
        // the field has been referenced
        $s'(\ell)(p) := s(\ell)(p)$
      end if
    end for
  end if
  // propagate the resulting state into the function entry
  $a.propagate'(c_2, entry(f_2), s')$
  // propagate flow for the return edge, if any is known already
  let $t = m(c_2, exit(f_2))$
  if $t \neq$ none **then**
    $a.funexit'(c_1, n_1, c_2, f_2, t)$
  end if

Fig. 4. Algorithm for $funentry'(c_1, n_1, c_2, f_2, s)$. This modified version of *funentry* "lazily" propagates s into the abstract state at $entry(f_2)$ in context c_2. The abstract state s' is unknown for all fields that have not yet been referenced by the function being called according to u (recall that $\bot_{\mathsf{State}}$ maps all fields to unknown).

$a.funexit'(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  let $(_, g) = a$ **and** $u_g = g(c_1, n_1, c_2, f_2)$
  $s' := \bot_{\mathsf{State}}$
  for all $\ell \in L, p \in P$ **do**
    if $s(\ell)(p) =$ unknown **then**
      // the field has not been accessed, so restore its value from the call edge state
      $s'(\ell)(p) := u_g(\ell)(p)$
    else
      $s'(\ell)(p) := s(\ell)(p)$
    end if
  end for
  $a.propagate'(c_1, after(n_1), s')$

Fig. 5. Algorithm for $funexit'(c_1, n_1, c_2, f_2, s)$. This modified version of *funexit* restores field values that have not been accessed within the function being called, using the value from before the call. It then propagates the resulting state as in the original operation.

Figure 6 illustrates the dataflow at function entries and exits as modeled by the $funexit'$ and $funentry'$ operations. The two nodes n_1 and n_2 represent function call statements that invoke the function f. Assume that the value of the field p in the abstract object ℓ, denoted $\ell.p$, is v_1 at n_1 and v_2 at n_2 where $v_1, v_2 \in \mathsf{Value}$. When dataflow first arrives at $entry(f)$ the $funentry'$ operation sets $\ell.p$ to unknown. Assuming that f does not access $\ell.p$ it remains unknown

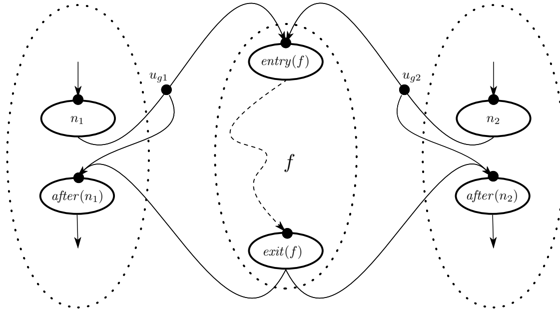


Fig. 6. A function f being called from two different statements, n_1 and n_2 appearing in other functions (for simplicity, all with the same context c). The edges indicate dataflow, and each bullet corresponds to an element of State with $u_{g1} = g(c, n_1, c, f)$ and $u_{g2} = g(c, n_2, c, f)$ where $g \in CallGraph$.

throughout f, so $funexit'$ can safely restore the original value v_1 by merging the state from $exit(f)$ with u_{g1} (the state recorded at the call edge) at $after(n_1)$. Similarly for the other call site, the value v_2 will be restored at $after(n_2)$. Thus, the dataflow for non-referenced fields respects the interprocedurally valid paths. This is in contrast to the basic framework where the value of $\ell.p$ would be $v_1 \sqcup v_2$ at both $after(n_1)$ and $after(n_2)$. Thereby, the modification of *funexit* may – perhaps surprisingly – cause the resulting analysis solution to be more precise than in the basic framework even for non-unknown field values. If a statement in f writes a value v' to $\ell.p$ it will no longer be unknown, so v' will propagate to both $after(n_1)$ and $after(n_2)$. If the transfer function of a statement in f invokes $getfield'$ to obtain the value of $\ell.p$ while it is unknown, it will be recovered by considering the call edges into f, as explained in Section 3.3.

The *getstate* operation is not modified. A transfer function cannot notice the fact that the returned State elements may contain unknown field values, because it is not permitted to read a field value through such a state.

Finally, the *getcallgraph* operation requires a minor modification to ensure that its output has the same type although the underlying lattice has changed:

$a.getcallgraph'()$:
  return $\{(c_1, n_1, c_2, f_2) \mid g(c_1, n_1, c_2, f_2) \neq \mathsf{none}\}$ **where** $(_, g) = a$

To demonstrate how the lazy propagation framework manages to avoid certain redundant computations, consider again the `markAsRunnable` function in Section 2.5. Suppose that the analysis first encounters a call to this function with some abstract state s. This call triggers the analysis of the function body, which accesses only a few object fields within s. The abstract state at the entry location of the function is unknown for all other fields. If new flow subsequently arrives via a call to the function with another abstract state s' where $s \sqsubseteq s'$, the introduction of unknown values ensures that the function body is only reanalyzed if s' differs from s at the few relevant fields that are not unknown.

3.3 Recovering Unknown Field Values

We now turn to the definition of the auxiliary operation *recover*. It gets invoked by *getfield'* and *propagate'* whenever an unknown element needs to be replaced by a proper field value. The operation returns the desired field value but also, as a side effect, modifies the relevant abstract states for function entry locations in a.

The key observation for defining $\mathit{recover}(c, n, \ell, p)$ where $c \in C$, $n \in N$, $\ell \in L$, and $p \in P$ is that unknown is only introduced in *funentry'* and that each call edge – very conveniently – records the abstract state just before the ordinary field value is changed into unknown. Thus, the operation needs to go back through the call graph and recover the missing information. However, it only needs to modify the abstract states that belong to function entry statements.

Recovery is a two phase process. The first phase constructs a directed multi-rooted graph G the nodes of which are a subset of $C \times F$. It is constructed from the call graph in a backward manner starting from (c, n) as the smallest graph satisfying the following two constraints, where $(m, g) = a$:

- If $u(\ell)(p) = \mathsf{unknown}$ where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$ then G contains the node $(c, \mathit{fun}(n))$.
- For each node (c_2, f_2) in G and for each (c_1, n_1) where $g(c_1, n_1, c_2, f_2) \neq \mathsf{none}$:
  - If $u_g(\ell)(p) = \mathsf{unknown} \wedge u_1(\ell)(p) = \mathsf{unknown}$ where $u_g = g(c_1, n_1, c_2, f_2)$ and $u_1 = m(c_1, \mathit{entry}(\mathit{fun}(n_1)))$ then G contains the node $(c_1, \mathit{fun}(n_1))$ with an edge to (c_2, f_2),
  - otherwise, (c_2, f_2) is a root of G.

The resulting graph is essentially a subgraph of the call graph such that every node (c', f') in G satisfies $u(\ell)(p) = \mathsf{unknown}$ where $u = m(c', \mathit{entry}(f'))$. A node is a root if at least one of its incoming edges contributes with a non-unknown value. Notice that root nodes may have incoming edges.

The second phase is a fixpoint computation over G:

```
// recover the abstract value at the roots of G
for each root (c', f') of G do
  let u' = m(c', entry(f'))
  for all (c1, n1) where g(c1, n1, c', f') ≠ none do
    let ug = g(c1, n1, c', f') and u1 = m(c1, entry(fun(n1)))
    if ug(ℓ)(p) ≠ unknown then
      u'(ℓ)(p) := u'(ℓ)(p) ⊔ ug(ℓ)(p)
    else if u1(ℓ)(p) ≠ unknown then
      u'(ℓ)(p) := u'(ℓ)(p) ⊔ u1(ℓ)(p)
    end if
  end for
end for
// propagate throughout G at function entry nodes
S := the set of roots of G
while S ≠ ∅ do
  select and remove (c', f') from S
```

```
    let u' = m(c', entry(f'))
    for each successor (c2, f2) of (c', f') in G do
      let u2 = m(c2, entry(f2))
      if u'(ℓ)(p) ⋢ u2(ℓ)(p) then
        u2(ℓ)(p) := u2(ℓ)(p) ⊔ u'(ℓ)(p)
        add (c2, f2) to S
      end if
    end for
  end while
```

This phase recovers the abstract value at the roots of G and then propagates the value through the nodes of G until a fixpoint is reached. Although *recover* modifies abstract states in this phase, it does not modify the worklist. After this phase, we have $u(\ell)(p) \neq$ unknown where $u = m(c', \mathit{entry}(f'))$ for each node (c', f') in G. (Notice that the side effects on a only concern abstract states at function entry statements.) In particular, this holds for $(c, \mathit{fun}(n))$, so when $\mathit{recover}(c, n, \ell, p)$ has completed the two phases, it returns the desired value $u(\ell)(p)$ where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$.

Notice that the graph G is empty if $u(\ell)(p) \neq$ unknown where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$ (see the first of the two constraints defining G). In this case, the desired field has already been recovered, the second phase is effectively skipped, and $u(\ell)(p)$ is returned immediately.

Figure 7 illustrates an example of interprocedural dataflow among four functions. (This example ignores dataflow for function returns and assumes a fixed calling context c.) The statements write_1 and write_2 write to a field $\ell.p$, and *read* reads from it. Assume that the analysis discovers all the call edges before visiting *read*. In that case, $\ell.p$ will have the value unknown when entering f_2 and f_3, which will propagate to f_4. The transfer function for *read* will then invoke $\mathit{getfield}'$, which in turn invokes *recover*. The graph G will be constructed with three nodes: (c, f_2), (c, f_3), and (c, f_4) where (c, f_2) and (c, f_3) are roots and have edges to (c, f_4). The second phase of *recover* will replace the unknown value of $\ell.p$ at $\mathit{entry}(f_2)$ and $\mathit{entry}(f_2)$ by its proper value stored at the call edges and then propagate that value to $\mathit{entry}(f_3)$ and finally return it to $\mathit{getfield}'$. Notice that the value of $\ell.p$ at, for example, the call edges, remains unknown. However, if dataflow subsequently arrives via transfer functions of other statements, those unknown values may be replaced by ordinary values. Finally, note that this simple example does not require fixpoint iteration within *recover*, however that becomes necessary when call graphs contain cycles (resulting from programs with recursive function calls).

The modifications only concern the AnalysisLattice ADT, in terms of which all transfer functions of an analysis are defined. The transfer functions themselves are not changed. Although invocations of *recover* involve traversals of parts of the call graph, the main worklist algorithm (Figure 1) requires no modifications.

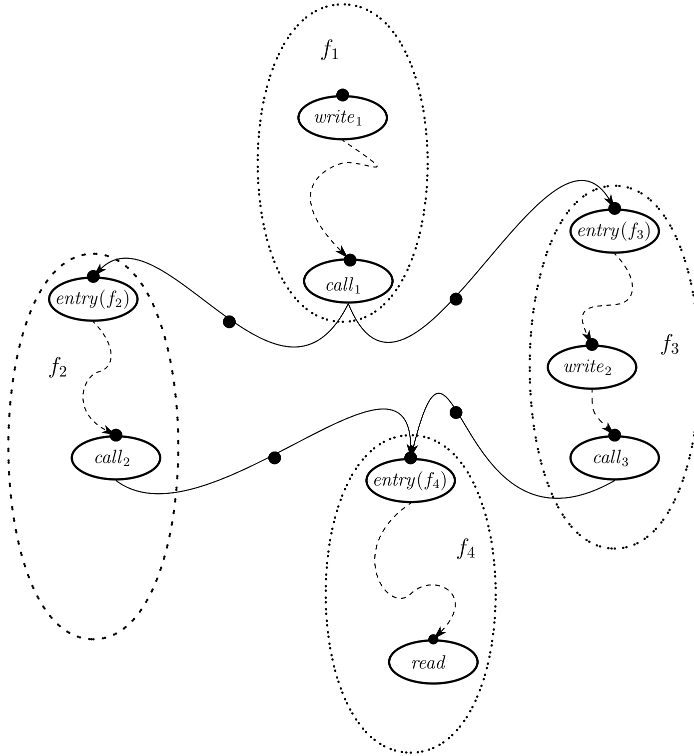


Fig. 7. Fragments of four functions, $f_1 \ldots f_4$. As in Figure 6, edges indicate dataflow and bullets correspond to elements of State. The statements $write_1$ and $write_2$ write to a field $\ell.p$, and *read* reads from it. The *recover* operation applied to the *read* statement and $\ell.p$ will ensure that values written at $write_1$ and $write_2$ will be read at the *read* statements, despite the possible presence of unknown values.

4 Implementation and Experiments

To examine the impact of lazy propagation on analysis performance, we extended the Java implementation of TAJS, our type analyzer for JavaScript [14], by systematically applying the modifications described in Section 3. As usual in dataflow analysis, primitive statements are grouped into basic blocks. The implementation focuses on the JavaScript language itself and the built-in library, but presently excludes the DOM API, so we use the most complex benchmarks from the V8[1] and SunSpider[2] benchmark collections for the experiments.

Descriptions of other aspects of TAJS not directly related to lazy propagation may be found in the TAJS paper [14]. These include the use of recency

[1] `http://v8.googlecode.com/svn/data/benchmarks/v1/`

[2] `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`

Table 1. Performance benchmark results

	LOC	Blocks	Iterations			Time (seconds)			Memory (MB)		
			basic	*basic+*	*lazy*	*basic*	*basic+*	*lazy*	*basic*	*basic+*	*lazy*
`richards.js`	529	478	2663	2782	1399	5.6	4.6	3.8	11.05	6.4	3.7
`benchpress.js`	463	710	18060	12581	5097	33.2	13.4	5.4	42.02	24.0	7.8
`delta-blue.js`	853	1054	∞	∞	63611	∞	∞	136.7	∞	∞	140.5
`cryptobench.js`	1736	2857	∞	43848	17213	∞	99.4	22.1	∞	127.9	42.8
`3d-cube.js`	342	545	7116	4147	2009	14.1	5.3	4.0	18.4	10.6	6.2
`3d-raytrace.js`	446	575	∞	30323	6749	∞	24.8	8.2	∞	16.7	10.1
`crypto-md5.js`	296	392	5358	1004	646	4.5	2.0	1.8	6.1	3.6	2.7
`access-nbody.js`	179	149	551	523	317	1.8	1.3	1.0	3.2	1.7	0.9

abstraction [4], which complicates the implementation, but does not change the properties of the lazy propagation technique.

We compare three versions of the analysis: *basic* corresponds to the basic framework described in Section 2; *basic+* extends the basic version with the copy-on-write and maybe-modified techniques discussed in Section 2.5, which is the version used in [14]; and *lazy* is the new implementation using lazy propagation (without the other extensions from the *basic+* version).

Table 1 shows for each program, the number of lines of code, the number of basic blocks, the number of fixpoint iterations for the worklist algorithm (Figure 1), analysis time (in seconds, running on a 3.2GHz PC), and memory consumption. We use ∞ to denote runs that require more than 512MB of memory.

We focus on the time and space requirements for these experiments. Regarding precision, *lazy* is in principle more precise than *basic+*, which is more precise than *basic*. On these benchmark programs, however, the precision improvement is insignificant with respect to the number of potential type related bugs, which is the precision measure we have used in our previous work.

The experiments demonstrate that although the copy-on-write and maybe-modified techniques have a significant positive effect on the resource requirements, lazy propagation leads to even better results. The results for `richards.js` are a bit unusual as it takes more iterations in *basic+* than in *basic*, however the fixpoint is more precise in *basic+*.

The benchmark results demonstrate that lazy propagation results in a significant reduction of analysis time without sacrificing precision. Memory consumption is reduced by propagating less information during the fixpoint computation and fixpoints are reached in fewer iterations by eliminating a cause of redundant computation observed in the basic framework.

5 Related Work

Recently, JavaScript and other scripting languages have come into the focus of research on static program analysis, partly because of their challenging dynamic nature. These works range from analysis for security vulnerabilities [29, 8] to static type inference [7, 27, 1, 14]. We concentrate on the latter category, aiming to develop program analyses that can compensate for the lack of static type

checking in these languages. The interplay of language features of JavaScript, including first-class functions, objects with modifiable prototype chains, and implicit type coercions, makes analysis a demanding task.

The IFDS framework by Reps, Horwitz, and Sagiv [23] is a powerful and widely used approach for obtaining precise interprocedural analyses. It requires the underlying lattice to be a powerset and the transfer functions to be distributive. Unfortunately, these requirements are not met by our type analysis problem for dynamic object-oriented scripting languages. The more general IDE framework also requires distributive transfer functions [25]. A connection to our approach is that fields that are marked as unknown at function exits, and hence have not been referenced within the function, are recovered from the call site in the same way local variables are treated in IFDS.

Sharir and Pnueli's functional approach to interprocedural analysis can be phrased both with symbolic representations and in an iterative style [26], where the latter is closer to our approach. With the complex lattices and transfer functions that appear to be necessary in analyses for object-oriented scripting languages, symbolic representations are difficult to work with, so TAJS uses the iterative style and a relatively direct representation of lattice elements. Furthermore, the functional approach is expensive if the analysis lattice is large.

Our analysis framework encompasses a general notion of context sensitivity through the C component of the analysis instances. Different instantiations of C lead to different kinds of context sensitivity, including variations of the call-string approach [26], which may also affect the quality of interprocedural analysis. We leave the choice of C open here; TAJS currently uses a heuristic that distinguishes call sites that have different values of `this`.

The introduction of unknown field values subsumes the *maybe-modified* technique that we used in the first version of TAJS [14]: a field whose value is unknown is definitely not modified. Both ideas can be viewed as instances of side effect analysis. Unlike, for example, the side effect analysis by Landi et al. [24] our analysis computes the call graph on-the-fly and we exploit the information that certain fields are found to be non-referenced for obtaining the lazy propagation mechanism. Via this connection to side effect analysis, one may also view the unknown field values as establishing a frame condition as in separation logic [21].

Combining call graph construction with other analyses is common in pointer alias analysis with function pointers, for example in the work of Burke et al. [11]. That paper also describes an approach called deferred evaluation for increasing analysis efficiency, which is specialized to flow insensitive alias analysis.

Lazy propagation is related to lazy evaluation (e.g., [22]) as it produces values passed to functions on demand, but there are some differences. Lazy propagation does not defer evaluation as such, but just the propagation of the values; it applies not just to the parameters but to the entire state; and it restricts laziness to data structures (values of fields).

Lazy propagation is different from demand-driven analysis [13]. Both approaches defer computation, but demand-driven analysis only computes results for selected hot spots, whereas our goal is a whole-program analysis that infers

information for all program points. Other techniques for reducing the amount of redundant computation in fixpoint solvers is difference propagation [6] and use of interprocedural def-use chains [28]. It might be possible to combine those techniques with lazy propagation, although they are difficult to apply to the complex transfer functions that we have in type analysis for JavaScript.

6 Conclusion

We have presented *lazy propagation* as a technique for improving the performance of interprocedural analysis in situations where existing methods, such as IFDS or the functional approach, do not apply. The technique is described by a systematic modification of a basic iterative framework. Through an implementation that performs type analysis for JavaScript we have demonstrated that it can significantly reduce the memory usage and the number of fixpoint iterations without sacrificing analysis precision. The result is a step toward sound, precise, and fast static analysis for object-oriented languages in general and scripting languages in particular.

Acknowledgments. The authors thank Stephen Fink, Michael Hind, and Thomas Reps for their inspiring comments on early versions of this paper.

References

1. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005)
2. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A.M., Ernst, M.D.: Finding bugs in dynamic web applications. In: Proc. International Symposium on Software Testing and Analysis, ISSTA 2008. ACM, New York (July2008)
3. Atkinson, D.C., Griswold, W.G.: Implementation techniques for efficient data-flow analysis of large programs. In: Proc. International Conference on Software Maintenance, ICSM 2001, pp. 52–61 (November 2001)
4. Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
5. Chase, D.R., Wegman, M., Kenneth Zadeck, F.: Analysis of pointers and structures. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1990 (June 1990)
6. Fecht, C., Seidl, H.: Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, p. 90. Springer, Heidelberg (1998)
7. Furr, M., An, Jong hoon (David), Foster, J.S., Hicks, M.W.: Static type inference for Ruby. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, Springer, Heidelberg (2009)

8. Guha, A., Krishnamurthi, S., Jim, T.: Using static analysis for Ajax intrusion detection. In: Proc. 18th International Conference on World Wide Web, WWW 2009 (2009)
9. Heidegger, P., Thiemann, P.: Recency types for analyzing scripting languages. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 200–224. Springer, Heidelberg (2010)
10. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2001, pp. 54–61 (June 2001)
11. Hind, M., Burke, M.G., Carini, P.R., Choi, J.-D.: Interprocedural pointer alias analysis. ACM Transactions on Programming Languages and Systems 21(4), 848–894 (1999)
12. Horwitz, S., Demers, A., Teitebaum, T.: An efficient general iterative algorithm for dataflow analysis. Acta Informatica 24(6), 679–694 (1987)
13. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE 1995 (October 1995)
14. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
15. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. Technical report, Department of Computer Science, Aarhus University (2010), http://cs.au.dk/~amoeller/papers/lazy/
16. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Proc. 9th ACM Symposium on Principles of Programming Languages, POPL 1982 (January 1982)
17. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. Journal of the ACM 23(1), 158–171 (1976)
18. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7, 305–317 (1977)
19. Kildall, G.A.: A unified approach to global program optimization. In: Proc. 1st ACM Symposium on Principles of Programming Languages. In: POPL 1973 (October 1973)
20. Liskov, B., Zilles, S.N.: Programming with abstract data types. ACM SIGPLAN Notices 9(4), 50–59 (1974)
21. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
22. Jones, S.L.P.: The Implementation of Functional Programming Languages. Prentice Hall, Englewood Cliffs (1987)
23. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 49–61 (January 1995)
24. Ryder, B.G., Landi, W., Stocks, P., Zhang, S., Altucher, R.: A schema for interprocedural modification side-effect analysis with pointer aliasing. ACM Transactions on Programming Languages and Systems 23(2), 105–186 (2001)

25. Sagiv, S., Reps, T.W., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science 167(1&2), 131–170 (1996)
26. Sharir, M., Pnueli, A.: Two approaches to interprocedural dataflow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–233. Prentice-Hall, Englewood Cliffs (1981)
27. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)
28. Tok, T.B., Guyer, S.Z., Lin, C.: Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 17–31. Springer, Heidelberg (2006)
29. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: Proc. 15th USENIX Security Symposium (August 2006)

Verifying a Local Generic Solver in Coq

Martin Hofmann[1], Aleksandr Karbyshev[2], and Helmut Seidl[2]

[1] Institut für Informatik, Universität München
hofmann@ifi.lmu.de
[2] Fakultät für Informatik, Technische Universität München
{aleksandr.karbyshev,seidl}@in.tum.de

Abstract. Fixpoint engines are the core components of program analysis tools and compilers. If these tools are to be trusted, special attention should be paid also to the correctness of such solvers. In this paper we consider the local generic fixpoint solver **RLD** which can be applied to constraint systems $\mathbf{x} \sqsupseteq f_{\mathbf{x}}, \mathbf{x} \in V$, over some lattice $\mathbb{D}$ where the right-hand sides $f_{\mathbf{x}}$ are given as arbitrary functions implemented in some specification language. The verification of this algorithm is challenging, because it uses higher-order functions and relies on side effects to track variable dependences as they are encountered dynamically during fixpoint iterations. Here, we present a correctness proof of this algorithm which has been formalized by means of the interactive proof assistant Coq.

1 Introduction

A *generic* solver computes a solution of a constraint system $\mathbf{x} \sqsupseteq f_{\mathbf{x}}, \mathbf{x} \in V$, over some lattice $\mathbb{D}$, where the right-hand side $f_{\mathbf{x}}$ of each variable $\mathbf{x}$ is given as a function of type $(V \to D) \to D$ implemented in some programming language. A *local* generic solver, when started with a set $X \subseteq V$ of *interesting* variables, tries to determine the values for the X of a solution of the constraint system by touching as few variables as possible.

Local generic solvers are a convenient tool for the implementation of efficient frameworks for program analyses. They have first been proposed for the analysis of logic programs [3,5,6,7] and model-checking [10], but recently have also attracted attention in interprocedural analyzers of imperative programs [1,14]. One particularly simple instance **RLD** of a local generic solver has been included into the textbook on *Program Analysis and Optimization* [15], although without any proof of correctness of the algorithm.

Efficient solvers for constraint systems exploit that often right-hand side functions query the current variable assignment only for few variables. A generic solver, however, must consider right-hand sides as *black boxes* which cannot be preprocessed for variable dependences before-hand. Therefore, efficient generic solvers rely on *self-observation* to detect and record variable dependences on-the-fly during evaluation of right-hand sides. The local generic solver **TD** by van Hentenryck [3] as well as the solver **RLD** add a recursive descent into solving

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 340–355, 2010.
© Springer-Verlag Berlin Heidelberg 2010

variables before reporting their values. Both self-observation through side-effects and the recursive evaluation make these solvers intricate in their operational behavior and therefore their design and implementation are error-prone.

In fact, during experimentation with tiny variations of the solver **RLD** we found that many seemingly correct algorithms and implementations are bogus. In view of the application of such solvers in tools for deriving correctness properties, possibly of safety critical systems, it seems mandatory to us to have full confidence into the applied software.

The first issue in proving any generic solver correct is which kind of functions safely may be applied as right-hand sides of constraints. In the companion paper [8] we therefore have presented a semantical property of *purity*. The notion of purity is general enough to allow any function expressed in a pure functional language without recursion, but also allows certain forms of (well-behaved) stateful computation. Purity of a function f allows f to be represented as a *strategy tree*. This means that the evaluation of f on a variable assignment σ can be considered as a sequence of variable look-ups followed by local computations and ending in an answer value.

It is w.r.t. this representation that we prove the local generic solver **RLD** correct. Related formal correctness proofs have been provided for variants of Kildall's algorithm for dataflow analysis [13,11,2,4] This fixpoint algorithm is neither generic nor local. It also exploits variable dependences which, however, are explicitly given through the control-flow graph.

All theorems and proofs are formalized by means of the interactive theorem prover Coq [12].

2 The Local Generic Solver RLD

One basic idea of the algorithm **RLD** is that, as soon as the value of variable $\mathbf{y}$ is requested during reevaluation of the right-hand side $f_{\mathbf{x}}$, the algorithm does not naively return the current value for $\mathbf{y}$. Instead, it first tries to get a better approximation of it, thus reducing the overall number of iterations and computations performed. This idea is similar to that of the algorithm **TD**.

Both algorithms also record the variable dependencies $(\mathbf{x}, \mathbf{y})$ (w.r.t. the current variable assignment) as they are encountered during evaluation of the right-hand side $f_{\mathbf{x}}$ as a *side-effect*. The main difference between the two algorithms lies in how they behave when a variable $\mathbf{x}$ changes its value. While the algorithm **TD** recursively *destabilizes* all variables which also indirectly depend on $\mathbf{x}$, the algorithm **RLD** only destabilizes the variables which immediately (locally) are influenced by $\mathbf{x}$, and triggers the reevaluation of these variables at once.

The algorithm **RLD** maintains the following data structures.

1. Finite map σ, storing current values of variables. We track only finite number of observed variables, since the overall size of set V can be tremendously large. We define the auxiliary function

$$\sigma_\perp\,\mathbf{x} = \begin{cases} \sigma\,\mathbf{x} & \text{if } \mathbf{x} \in \mathrm{dom}(\sigma), \\ \perp & \text{otherwise} \end{cases}$$

that returns a current value of $\sigma\,\mathbf{x}$ if it is defined; otherwise, it returns $\perp$.

2. Finite set *stable* $\subseteq V$. Intuitively, if variable $\mathbf{x}$ is marked as stable then either $\mathbf{x}$ is already *solved*, i.e., a computation for $\mathbf{x}$ has completed and σ gives a solution for $\mathbf{x}$ and all those variables $\mathbf{x}$ transitively depends on, or $\mathbf{x}$ is *called* and it is in the call stack of `solve` function and its value is being processed.
3. Finite map *infl*, where dependencies between variables are stored. More exactly, *infl* $\mathbf{x}$ returns an overapproximation of a set of variables $\mathbf{y}$, for which evaluation of $f_\mathbf{y}$ on the current $\sigma_\perp$ depends on $\mathbf{x}$. Again, we track only finite number of observed variables and define the auxiliary function

$$\mathit{infl}_{[\,]}\,\mathbf{x} = \begin{cases} \mathit{infl}\,\mathbf{x} & \text{if } \mathbf{x} \in \mathrm{dom}(\mathit{infl}), \\ [\,] & \text{otherwise.} \end{cases}$$

The structures have initial values: $\sigma = \emptyset$, $\mathit{stable} = \emptyset$, $\mathit{infl} = \emptyset$.

The algorithm **RLD** proceeds as follows (see Fig. 1). The function `solve_all` is called for a list X of interesting variables from the initial state (with $\sigma = \emptyset$, $\mathit{stable} = \emptyset$, $\mathit{infl} = \emptyset$). The function `solve_all` calls recursively `solve` $\mathbf{x}$ for every $\mathbf{x} \in X$.

The function `solve` when called for some variable $\mathbf{x}$ first checks whether $\mathbf{x}$ is already in the set *stable*. If so, the function returns; otherwise, the algorithm marks $\mathbf{x}$ as being stable and tries to satisfy a constraint $\sigma\,\mathbf{x} \sqsupseteq f_\mathbf{x}\,\sigma$. For that, it reevaluates a value of the right-hand side $f_\mathbf{x}$, and calculates the least upper bound *new* of the result together with the old value of $\sigma\,\mathbf{x}$. If the value of *new* is strictly larger than the old value, the function `solve` updates the value of σ for $\mathbf{x}$. Since the value of $\sigma\,\mathbf{x}$ has changed, all constraints of variables $\mathbf{y}$ dependent on $\mathbf{x}$ may not be satisfied anymore. Hence the function `solve` *destabilizes* all the variables from $\mathit{work} = \mathit{infl}_{[\,]}\,\mathbf{x}$, i.e., subtracts *work* from the set *stable*. Then value $\mathit{infl}\,\mathbf{x}$ is reset to empty and `solve_all` *work* is recursively called.

We mention, that the right-hand side $f_\mathbf{x}$ is not evaluated *directly* on the function σ, but by using an auxiliary stateful function $\lambda y.$`eval(`$\mathbf{x}$,y`)`, allowing firstly to get better values for variables the variable $\mathbf{x}$ depends on. Once `eval(`$\mathbf{x}$,$\mathbf{y}$`)` is called, it first calls `solve` $\mathbf{y}$ and then adds $\mathbf{x}$ to $\mathit{infl}\,\mathbf{y}$. The latter reflects the fact that the value of $\mathbf{x}$ possibly depends on the value of $\mathbf{y}$. Only after recording the variable dependence $(\mathbf{x}, \mathbf{y})$, the current value of $\mathbf{y}$ is returned.

Our goal is to prove that the algorithm **RLD** is a local generic solver for any (possibly infinite) constraint system $\mathcal{S} = (V, f)$ where right-hand sides $f_\mathbf{x}$ are *pure*.

3 Systems of Constraints

Instead of reasoning about an algorithm which modifies a global state by side-effecting functions as in Fig. 1, we prefer to reason about the *denotational semantics* of such an algorithm, i.e., about the corresponding purely functional program where the global state is explicitly threaded through the program.

```
function eval(x : V, y : V) =
    solve(y);
    infl y ← infl y ∪ {x};
    σ⊥ y
function eval_rhs(x : V) =
    f_x(λy.eval(x, y))

function extract_work(x : V) =
    let work = infl_[] x in
    stable ← stable \ work; infl x ← [];
    work
function solve(x : V) =
    if x ∈ stable then ()
    else
        stable ← stable ∪ {x};
        let cur = eval_rhs(x) in
        let new = σ⊥ x ⊔ cur in
        if new ⊑ σ⊥ x then ()
        else
            σ x ← new;
            let work = extract_work(x) in
            solve_all(work)
        end
    end
function solve_all(work : 2^V) =
    foreach x ∈ work do solve(x)

begin
    σ = ∅; stable = ∅; infl = ∅;
    solve_all(X);
    (σ⊥, stable)
end
```

Fig. 1. The recursive solver tracking local dependencies (**RLD**)

Assume $\mathbb{D} = (D, \sqcup, \sqsubseteq)$ is a lattice consisting of the carrier D equipped with the partial ordering $\sqsubseteq$ and the least upper bound operation $\sqcup$. A pair (V, f) is a *constraint system*, where V is a set of variables and f is a functional of type

$$f : V \to (V \to \mathcal{M}D) \to \mathcal{M}D,$$

that for every $\mathbf{x} \in V$ returns a corresponding *right-hand side* $f_{\mathbf{x}} : (V \to \mathcal{M}D) \to \mathcal{M}D$. Here, the monad constructor $\mathcal{M}$ when applied to a set D, returns a computation resulting in a value from D. In our application, we assume $\mathcal{M}D$ to be a *state transformer* monad defined by $S \to (D \times S)$ for some set S of states where f is assumed to be *polymorphic* in S.

This means that right-hand sides may have side effects onto the global state and that they can be applied to variable assignments whose evaluation themselves may have side effects. What we assume, however, is that the side effects

of the evaluation of a call $f_{\mathbf{x}}\,\sigma$ only are attributed to side-effects incurred by the evaluation of the function σ. This property is *not* captured by polymorphism in a state alone [8]. It is guaranteed, however, by the notion of *purity* introduced in [8]. If the function $f_{\mathbf{x}}$ is pure in the sense of [8], then $f_{\mathbf{x}}$ is representable by means of a *strategy tree*. This means that the evaluation of $f_{\mathbf{x}}$ on a variable assignment consists of a sequence of variable look-ups followed by some local computation leading to further look-ups and so on until eventually a result is produced.

3.1 Strategy Trees

Definition 1. *For a given set of values D and a set of variables V we define the set $\mathcal{T}(V, D)$ of* strategy trees *inductively by:*

- *if $a \in D$ then $\mathsf{Answ}(a) \in \mathcal{T}(V, D)$;*
- *if $\mathbf{x} \in V$ and $c : D \to \mathcal{T}(V, D)$ is a total function then $\mathsf{Quest}(\mathbf{x}, c) \in \mathcal{T}(V, D)$.*

Let τ be a mapping from $V \to \mathcal{M}D$. By means of the monad operations **return** : $D \to \mathcal{M}D$ *and* **bind** : $\mathcal{M}D \to (D \to \mathcal{M}D) \to \mathcal{M}D$ *we define the function*

$$[\![\cdot]\!] : \mathcal{T}(V, D) \to (V \to \mathcal{M}D) \to \mathcal{M}D$$

recursively by:

$$\begin{aligned}[\![\mathsf{Answ}(a)]\!]\,\tau &= \textbf{return}\ a\,,\\ [\![\mathsf{Quest}(\mathbf{x}, c)]\!]\,\tau &= \textbf{bind}\ (\tau\,\mathbf{x})\ (\textbf{fun}\ a \to [\![c\ a]\!]\,\tau)\,.\end{aligned}$$

Recall that for state transformer monads, the monad operations **return** : $D \to \mathcal{M}D$ and **bind** : $\mathcal{M}D \to (D \to \mathcal{M}D) \to \mathcal{M}D$ are defined by:

$$\begin{aligned}\textbf{return}\ a &= \textbf{fun}\ s \to (a, s)\,,\\ \textbf{bind}\ m\,f &= \textbf{fun}\ s \to \textbf{let}\ (a, s_1) = m\ s\ \textbf{in}\ f\,a\,s_1\,.\end{aligned}$$

Therefore, the function $[\![\cdot]\!]$ is given by:

$$\begin{aligned}[\![\mathsf{Answ}(a)]\!]\,\tau &= \textbf{fun}\ s \to (a, s)\,,\\ [\![\mathsf{Quest}(\mathbf{x}, c)]\!]\,\tau &= \textbf{fun}\ s \to \textbf{let}\ (a, s_1) = \tau\,\mathbf{x}\,s\ \textbf{in}\ [\![c\ a]\!]\,\tau\,s_1\,.\end{aligned}$$

The evaluation of a strategy tree thus formalizes the stateful evaluation of the pure function represented by this tree.

Moreover, if τ does not depend on the state and has no effect on the state, i.e., is of the form

$$\tau = \textbf{return} \circ \sigma = \textbf{fun}\ \mathbf{x} \to \textbf{return}\ (\sigma\,\mathbf{x})$$

for some function $\sigma : V \to D$, then for all states s and trees $r \in \mathcal{T}(V, D)$

$$[\![r]\!]\,\tau\,s = (a, s)$$

holds, for some $a \in D$. Therefore, we define the function

$$[\![\cdot]\!]^* : \mathcal{T}(V, D) \to (V \to D) \to D$$

by:

$$[\![r]\!]^* \, \sigma = \mathit{fst}([\![r]\!] \, (\mathbf{return} \circ \sigma) \, ()) \, .$$

In our application, the solver not only evaluates pure functions, i.e., strategy trees, but also records the variables accessed during this evaluation. In order to reason about the sequence of accessed variables together with their values, we *instrument* the evaluation of strategy trees by additionally taking a list of already visited variables together with their values and returning updated list for the rest computations. For the state transformer monad this instrumented evaluation is defined by:

$$\begin{aligned} [\![\mathsf{Answ}(a)]\!]' \, \tau \, l \quad &= \mathbf{return} \; (a, l) \, , \\ [\![\mathsf{Quest}(\mathbf{x}, c)]\!]' \, \tau \, l &= \mathbf{bind} \; (\tau \, \mathbf{x}) \; (\mathbf{fun} \; a \to [\![c \; a]\!]' \, \tau \, (l \, @ \, [(\mathbf{x}, a)])) \, , \end{aligned}$$

or, again instantiated for state transformer monads,

$$\begin{aligned} [\![\mathsf{Answ}(a)]\!]' \, \tau \, l \quad &= \mathbf{fun} \; s \to ((a, l), s) \, , \\ [\![\mathsf{Quest}(\mathbf{x}, c)]\!]' \, \tau \, l &= \mathbf{fun} \; s \to \mathbf{let} \; (a, s_1) = \tau \, \mathbf{x} \, s \; \mathbf{in} \; [\![c \; a]\!]' \, \tau \, (l \, @ \, [(\mathbf{x}, a)]) \, s_1 \, , \end{aligned}$$

where $l : (V \times D)$ *list*.

Then for every strategy tree r, mapping $\tau : V \to \mathcal{M}D$ and list $l_1 : (V \times D)$ *list*

$$[\![r]\!] \, \tau \, s = (a, s') \qquad \text{iff} \quad [\![r]\!]' \, \tau \, l_1 \, s = ((a, l_2), s') \, ,$$

for some $a \in D$ and $l_2 : (V \times D)$ *list*. Moreover, if $\tau = \mathbf{return} \circ \sigma$ for some $\sigma : V \to D$, then for all states s

$$[\![r]\!]' \, \tau \, [\,] \, s = ((a, l), s)$$

holds, for some $a \in D$ and $l : (V \times D)$ *list*.

Now assume that we are given a mapping $t : V \to \mathcal{T}(V, D)$. Relative to this mapping and an assignment $\sigma : V \to D$ we define

$$\begin{aligned} &\mathsf{trace}_\sigma \, r = l \, , \quad \text{where} \quad [\![r]\!]' \, (\mathbf{return} \circ \sigma) \, [\,] \, () = ((_, l), _), \; r \in \mathcal{T}(V, D) \, , \\ &\mathsf{dep}_{t,\sigma} \, \mathbf{x} = \{\mathbf{y} \mid (\mathbf{y}, _) \in \mathsf{trace}_\sigma(t \; \mathbf{x})\} \, . \end{aligned}$$

Moreover, we define $\mathsf{dep}_{t,\sigma}(X) = \bigcup_{\mathbf{x} \in X} \mathsf{dep}_{t,\sigma} \, \mathbf{x}$. Intuitively, the function $\mathsf{dep}_{t,\sigma}$ applied to a variable $\mathbf{x}$ and a variable assignment σ returns a set of variables that $\mathbf{x}$ *directly depends on relative to* σ, i.e., a set of those variables which values are required to evaluate the strategy tree for the right-hand side of $\mathbf{x}$. The relation

$$\mathsf{Dep}_{t,\sigma} = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in \mathsf{dep}_{t,\sigma} \, \mathbf{x}\}$$

is also called a *dependence graph* for the variable assignment σ. Let $\mathsf{Dep}^+_{t,\sigma}$ be a transitive closure of the relation $\mathsf{Dep}_{t,\sigma}$ and $\mathsf{Dep}^*_{t,\sigma} = \mathsf{Dep}^+_{t,\sigma} \cup \{(\mathbf{x}, \mathbf{x}) \mid \mathbf{x} \in V\}$ be a reflexive and transitive closure of $\mathsf{Dep}_{t,\sigma}$ and denote $\mathsf{dep}^*_{t,\sigma} \, \mathbf{x} = \{\mathbf{y} \mid \mathsf{Dep}^*_{t,\sigma}(\mathbf{x}, \mathbf{y})\}$ and $\mathsf{dep}^*_{t,\sigma}(X) = \bigcup_{\mathbf{x} \in X} \mathsf{dep}^*_{t,\sigma} \, \mathbf{x}$.

3.2 Solutions

Definition 2. *Let $\mathcal{S} = (V, f)$ be a constraint system over the lattice $\mathbb{D}$ and $X \subseteq V$. We say that a variable assignment $\sigma : V \to D$ is a* solution *of the constraint system $\mathcal{S}$, if for every $\mathbf{x} \in V$, $\sigma\,\mathbf{x} \sqsupseteq d$ whenever $(d, ()) = f_{\mathbf{x}}(\mathbf{return} \circ \sigma)\,()$ holds. For the latter statement, we also write $\sigma\,\mathbf{x} \sqsupseteq f_{\mathbf{x}}\,\sigma$.*

Definition 3. *A partial function*

$$\mathcal{A} : (V \to \mathcal{T}(V, D)) \times 2^V \to (V \to D) \times 2^V$$

is (the denotational semantics of) a local solver *if it takes as input a pair (t, X) of a strategy function t and a set $X \subseteq V$ of interesting variables and, whenever it terminates, returns a pair (σ, X') consisting of a variable assignment $\sigma : V \to D$ together with a set $X' \subseteq V$ such that the following holds:*

1. *$X \subseteq X'$ and $\mathsf{dep}^*_{t,\sigma}(X') \subseteq X'$;*
2. *$\sigma\,\mathbf{x} \sqsupseteq [\![t\,\mathbf{x}]\!]^*\,\sigma$ holds for every $\mathbf{x} \in X'$.*

In particular, this means that σ restricted to X' is a solution of the constraint system $(X', f\,|_{X'})$.

4 Functional Implementation with Explicit State Passing

In the functional implementation of algorithm **RLD**, the global state is made explicit, and passed into function calls by means of a separate parameter. Accordingly, the modified state together with the computed value (if there is any) are jointly returned. The type of a state is

$$\mathbf{type}\ \mathsf{state} = 2^V \times (V \rightharpoonup D) \times (V \rightharpoonup V\ \mathit{list})\,.$$

The three components correspond to the set *stable*, the finite (partial) map σ, and the finite (partial) map *infl*, respectively.

To facilitate the handling of the state we introduce the following auxiliary functions:

- The function `get` : $\mathsf{state} \to V \to D$ implements the function $\sigma_\bot$;
- The function `set` : $V \to D \to \mathsf{state} \to \mathsf{state}$ when applied to $\mathbf{x}$ updates the current value of $\sigma\,\mathbf{x}$;
- The function `get_stable` : $\mathsf{state} \to 2^V$ extracts the set *stable* from the current state;
- The function `is_stable` : $V \to \mathsf{state} \to \mathbf{bool}$ checks whether a given variable $\mathbf{x}$ is in the set *stable*;
- The function `add_stable` : $V \to \mathsf{state} \to \mathsf{state}$ adds a given variable to the set *stable*;
- The function `rem_stable` : $V \to \mathsf{state} \to \mathsf{state}$ removes a given variable from the set *stable*;
- The function `get_infl` : $V \to \mathsf{state} \to V\ \mathit{list}$ implements the function $\mathit{infl}_{[\,]}$;

```
let rec eval x y = fun s →
    let s = solve y s in
    let s = add_infl y x in
    (get y s, s)
and eval_rhs x = fun s →
    ⟦t x⟧ (eval x) s

and solve x = fun s →
    if is_stable x s then s
    else
        let s = add_stable x s in
        let (new_val, s) = eval_rhs x s in
        let cur_val = get s x in
        let new_val = cur_val ⊔ new_val in
        if new_val ⊑ cur_val then s
        else
            let s = set x new_val s in
            let (work, s) = extract_work x s in
            solve_all work s
and solve_all work = fun s →
    match work with
    | [] → s
    | x :: xs → solve_all xs (solve x s) in
let s_init = (∅, ∅, ∅) in
let s = solve_all X s_init in
(get s, get_stable s)
```

Fig. 2. Functional implementation of **RLD**

- The function `add_infl` : $V \to V \to \mathsf{state} \to \mathsf{state}$ applied to variables $\mathbf{x}$ and $\mathbf{y}$ adds the pair $(\mathbf{y}, \mathbf{x})$ to *infl*;
- The function `rem_infl` : $V \to \mathsf{state} \to \mathsf{state}$ applied to the variable $\mathbf{x}$ sets the list $\mathit{infl}_{[\,]}\ \mathbf{x}$ in the current state to $[\,]$.

The auxiliary function `extract_work` : $V \to \mathsf{state} \to (V\ \mathit{list} \times \mathsf{state})$ applied to a variable $\mathbf{x}$ determines the list w of variables immediately influenced by $\mathbf{x}$, resets $\mathit{infl}\,\mathbf{x}$ to $[\,]$, and subtracts w from the set *stable* as follows:

```
let extract_work x = fun s →
    let w = get_infl x s in
    let s = rem_infl x s in
    let s = fold_left (fun s y → rem_stable y s) s w in
    (w, s)
```

Using the auxiliary functions $[\![\cdot]\!]$ for strategy trees, the mutually recursive functions `eval`, `eval_rhs`, `solve` and `solve_all` of the algorithm are then given in Fig. 2.

Given a list of interesting variables $X \subseteq V$ the algorithm calls the function `solve_all` from the initial state $\mathsf{s_init} = (\emptyset, \emptyset, \emptyset)$.

From now on, **RLD** refers to this functional implementation. We prove:

Theorem 4. *The algorithm* **RLD** *is a local generic solver.*

5 Proof of Theorem 4

The proof consists of four main steps:

1. We instrument the functional program, introducing auxiliary data structures — ghost variables.
2. We implement the instrumented program in Coq.
3. We provide invariants for the instrumented program.
4. We prove these invariants jointly by induction on number of recursive calls.

5.1 Instrumentation

In order to express the invariants necessary to prove the correctness of the algorithm, we introduce additional components into the state which do not affect the operational behavior of the algorithm but record auxiliary information. The auxiliary data structures appear in the program as *ghost variables*, i.e., variables which are not allowed to appear in case distinctions and may not be written into ordinary structures. Thus, they do not influence the "control flow" of the program. We distinguish:

– the set *called* of variables which are currently processed;
– the set *queued* of variables which have been *destabilized*, i.e., removed from the set *stable* by the algorithm and not yet been reevaluated.

Accordingly, the type state in the instrumented program is given by:

$$\textbf{type}\ \mathsf{state} = 2^V \times 2^V \times (V \rightharpoonup D) \times (V \rightharpoonup V\ \mathit{list}) \times 2^V\ .$$

The five components correspond to the sets *stable* and *called*, the finite (partial) map σ, the finite (partial) map *infl*, and the set *queued*, respectively.

Also, we require the following auxiliary functions:

– The function `add_called` $: V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ adds a given variable to the set *called*;
– The function `rem_called` $: V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ removes a given variable from the set *called*;
– The function `add_queued` $: V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ adds a given variable to the set *queued*;
– The function `rem_queued` $: V \rightarrow \mathsf{state} \rightarrow \mathsf{state}$ removes a given variable from the set *queued*.

```
(*...*)
and eval_rhs x = fun s →
    ⟦t x⟧′ (eval x) [] s

and solve x = fun s →
    if is_stable x s then s
    else
        let s = rem_queued x s in
        let s = add_stable x s in
        let s = add_called x s in
        let ((new_val, _), s) = eval_rhs x s in
        let s = rem_called x s in
        let cur_val = get s x in
        let new_val = cur_val ⊔ new_val in
        if new_val ⊑ cur_val then s
        else
            let s = set x new_val s in
            let (work, s) = extract_work x s in
            solve_all work s
```

Fig. 3. Instrumented implementation of the functions `eval_rhs` and `solve`

In the instrumented implementation, we also replace the evaluation $[\![\cdot]\!]$ for strategy trees with $[\![\cdot]\!]'$ which additionally returns the list of accessed variables together with their respective values. Also, the function `extract_work` for a given **x** additionally removes the list w of variables influenced by **x** from the set *called* and adds it to the set *queued* of the current state.

The instrumented functions `eval_rhs` and `solve` are given in Fig. 3. The functions `eval` and `solve_all` remain unchanged.

It is intuitively clear that the instrumentation does not alter the relevant behavior of the algorithm and that therefore the subsequent verification of the instrumented version also establishes the correctness of the original one. We now sketch two ways for making this rigorous; neither of them is part of the formal verification, though, which operates entirely on the instrumented version. For the rest of this section let us used primed notation, e.g. state', $\mathtt{solve}'$ etc. for the instrumented versions, leaving the unprimed ones for the original version.

We can define a simulation relation $\sim \subseteq \mathsf{state} \times \mathsf{state}'$ as the graph of the projection from state' to state. We define a lifted relation $\mathcal{M}(\sim) \subseteq \mathcal{M}X \times \mathcal{M}'X$ for any X by

$$f \; \mathcal{M}(\sim) \; f' \equiv \forall s, s', s_1, s_1', x, x' . \, f(s) = (x, s_1) \wedge f'(s') = (x', s_1') \wedge \\ s \sim s' \implies s_1 \sim s_1' \wedge x = x' \, .$$

Two functions $f : X \to \mathcal{M}Y$ and $f' : X \to \mathcal{M}'Y$ are related if $f(x) \; \mathcal{M}(\sim) \; f'(x)$ holds for all $x \in X$. It is then a straightforward consequence from the definitions that each component of the algorithm is related to its primed (instrumented) version and thus that they yield equal results when started in related states and after discarding the instrumentation.

Alternatively, we can modify the verification of the instrumented version to yield a direct verification of the original version by existentially quantifying the instrumentation components in all invariants. When showing that such existentially quantified invariants are indeed preserved, one opens the existentials in the assumption yielding a fixed but arbitrary instrumentation of the starting state; one then updates this instrumentation using the above updating functions `rem_queued`, `add_stable` etc. and uses the resulting instrumentation as existential witness for the conclusion. The remaining proof obligation then follows step by step the verification of the instrumented version. See [9] for a formal account of this proof-transforming procedure in the context of Hoare logic.

5.2 Implementation in Coq

Coq accepts the definition of a recursive function only if it is provably terminating. Since the algorithm **RLD** is generic, we neither make any assumptions concerning the lattice $\mathbb{D}$ (e.g., w.r.t. finiteness of ascending chains), nor assume finiteness of the set of variables V. Accordingly, termination of the algorithm cannot be guaranteed. Therefore, our formalization of the algorithm in Coq relies on the representation of partial functions through their function graphs. The mutual recursive definition of these *relations* exactly mimics the functional implementation of the algorithm.

We define the following relations (see appendix):

- for every $\mathbf{x}, \mathbf{y} \in V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $d \in D$, $\mathtt{EvalRel}(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{s}', d)$ holds iff the call $\mathtt{eval}\,\mathbf{x}\,\mathbf{y}\,\mathbf{s}$ terminates and returns the value $(d, \mathbf{s}')$;
- for every $\mathbf{x} \in V$, $t \in \mathcal{T}(D,T)$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $d \in D$, $l, l' : (V \times D)\,\mathit{list}$, $\mathtt{Wrap_Eval_x}(\mathbf{x}, t, \mathbf{s}, \mathbf{s}', d, l, l')$ holds iff the call $[\![t]\!]'\,(\mathtt{eval}\ \mathbf{x})\ l\ \mathbf{s}$ terminates and returns the value $((d, l'), \mathbf{s}')$;
- for every $\mathbf{x} \in V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $d \in D$, $l' : (V \times D)\,\mathit{list}$, $\mathtt{Eval_rhs}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l')$ holds iff the call $\mathtt{eval_rhs}\,\mathbf{x}\,\mathbf{s}$ terminates and returns the value $((d, l'), \mathbf{s}')$;
- for every $\mathbf{x} \in V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $\mathtt{Solve}(\mathbf{x}, \mathbf{s}, \mathbf{s}')$ holds iff the call $\mathtt{solve}\ \mathbf{x}\ \mathbf{s}$ terminates and returns the value $\mathbf{s}'$;
- for every $\mathit{work} \subseteq V$, $\mathbf{s}, \mathbf{s}' : \mathsf{state}$, $\mathtt{SolveAll}(\mathit{work}, \mathbf{s}, \mathbf{s}')$ holds iff the call $\mathtt{solve_all}\ \mathit{work}\,\mathbf{s}$ terminates and returns the value $\mathbf{s}'$.

The defined predicates relate states before the call and after termination of the corresponding functions. Therefore, they can be used to reason about properties of the algorithm, even if its termination is not guaranteed.

5.3 Invariants

Given a variable assignment σ we inductively define relation $\mathsf{valid} \subseteq (V \times D)\,\mathit{list} \times (V \to D)$ as follows:

- $\mathsf{valid}([\,], \sigma)$;
- for any $\mathbf{x} \in V$, $d \in D$ and $l : (V \times D)\,\mathit{list}$, if $\mathsf{valid}(l, \sigma)$ and $d = \sigma\,\mathbf{x}$ then $\mathsf{valid}((\mathbf{x}, d) {::} l, \sigma)$;

and relation $\mathsf{legal} \subseteq (V \times D)\,\textit{list} \times \mathcal{T}(V, D)$ inductively by:

– $\mathsf{legal}([\,], r)$ for any $r \in \mathcal{T}(V, D)$;
– for any $\mathbf{x} \in V$, $d \in D$, $l : (V \times D)\,\textit{list}$ and $c : D \to \mathcal{T}(V, D)$, if $\mathsf{legal}(l, c(d))$ then $\mathsf{legal}((\mathbf{x}, d)::l, \mathsf{Quest}(\mathbf{x}, c))$.

Intuitively, $\mathsf{valid}(l, \sigma)$ holds iff the path l agrees with the variable assignment σ, and $\mathsf{legal}(l, r)$ means that one can walk along the path l in the tree r, for every $(\mathbf{x}, d)$ from l using a value d as an argument of a corresponding continuation function. For example, one can show by induction that $\mathsf{trace}_\sigma\, r$ is valid for σ and is legal in r, i.e., $\mathsf{valid}(\mathsf{trace}_\sigma\, r, \sigma)$ and $\mathsf{legal}(\mathsf{trace}_\sigma\, r, r)$ hold for any $r \in \mathcal{T}(V, D)$ and given variable assignment σ.

Given a strategy tree r and a path l legal in r we can define a function $\mathsf{subtree}(l, r)$ recursively as follows:

– if $l = [\,]$ then $\mathsf{subtree}(l, r) = r$;
– if $l = (\mathbf{x}, d)::vs$ and $r = \mathsf{Quest}(\mathbf{x}, c)$ then $\mathsf{subtree}(l, r) = \mathsf{subtree}(vs, c(d))$.

We have that $\mathsf{subtree}(\mathsf{trace}_\sigma\, r, r) = \mathsf{Answ}(a)$ holds for every tree $r \in \mathcal{T}(V, D)$ and variable assignment σ.

We prove by induction on length of a path the following lemma.

Lemma 5. *For any given $r \in \mathcal{T}(V, D)$, a path $l : (V \times D)\,\textit{list}$ and a variable assignment $\sigma : V \to D$, the following is equivalent:*

– $l = \mathsf{trace}_\sigma\, r$;
– $\mathsf{valid}(l, \sigma)$, $\mathsf{legal}(l, r)$, $\mathsf{subtree}(l, r) = \mathsf{Answ}(a)$, *for some $a \in D$, hold.* □

From now on, for simplicity, we denote `get_infl` as $\textit{infl}_{[\,]}$ and `get` as $\sigma_\bot$. States $\mathbf{s}$ and $\mathbf{s}'$ denote a state before a call of some function and a state after the call terminates, respectively. Structures *stable*, *called*, *queued* and *infl* are components of the state $\mathbf{s}$, primed structures are components of the state $\mathbf{s}'$. Let $t : V \to \mathcal{T}(V, D)$ be a given strategy function. We denote a tree $t\,\mathbf{x}$ by $t_\mathbf{x}$. We say that variable $\mathbf{x}$ is *solved* in the state $\mathbf{s}$ if $\mathbf{x} \in \textit{stable} \setminus \textit{called}$. We treat lists as sets in the formulae below.

We define:

$$\mathcal{I}_0(\mathbf{s}) \equiv \textit{called} \subseteq \textit{stable} \wedge \textit{queued} \cap \textit{stable} = \emptyset\,,$$
$$\mathcal{I}_1(\mathbf{s}, \mathbf{s}') \equiv \textit{stable}' \supseteq \textit{stable} \wedge \textit{called}' \subseteq \textit{called} \wedge \textit{queued}' \subseteq \textit{queued}\,.$$

We call a state $\mathbf{s}$ (a transition from $\mathbf{s}$ to $\mathbf{s}'$) *consistent* if $\mathcal{I}_0(\mathbf{s})$ (respectively, $\mathcal{I}_1(\mathbf{s}, \mathbf{s}')$) holds. The formula

$$\mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \equiv \forall \mathbf{z} \in V.\ \sigma_\bot\, \mathbf{s}'\, \mathbf{z} \sqsupseteq \sigma_\bot\, \mathbf{s}\, \mathbf{z}$$

expresses that the variable assignment in the state $\mathbf{s}'$ returns larger values than that in the state $\mathbf{s}$. The formula

$$\mathcal{I}_{\sigma,\textit{infl}}(\mathbf{s}, \mathbf{s}') \equiv \forall \mathbf{z} \in V.\, (\sigma_\bot\, \mathbf{s}'\, \mathbf{z} \sqsubseteq \sigma_\bot\, \mathbf{s}\, \mathbf{z} \implies \textit{infl}_{[\,]}\, \mathbf{z}\, \mathbf{s} \subseteq \textit{infl}_{[\,]}\, \mathbf{z}\, \mathbf{s}') \wedge$$
$$(\sigma_\bot\, \mathbf{s}'\, \mathbf{z} \not\sqsubseteq \sigma_\bot\, \mathbf{s}\, \mathbf{z} \implies \textit{infl}_{[\,]}\, \mathbf{z}\, \mathbf{s} \subseteq \textit{stable}' \setminus \textit{called}')$$

relates structures σ and *infl*. It expresses for every variable $\mathbf{z}$ the following. If the value of $\mathbf{z}$ did not increase, then *infl*$'$ contains more dependencies; otherwise, all the variables influenced by $\mathbf{z}$ in $\mathbf{s}$ are solved in $\mathbf{s}'$. The formula

$$\mathcal{I}_{\text{dep}}(\mathbf{x}, \mathbf{s}) \equiv \forall \mathbf{z} \in \mathsf{dep}_{t,(\sigma_\perp\,\mathbf{s})}\,\mathbf{x}.\, \mathbf{z} \in \mathit{stable} \cup \mathit{queued} \wedge \mathbf{x} \in \mathit{infl}_{[\,]}\,\mathbf{z}\,\mathbf{s}\,.$$

expresses that for every variable $\mathbf{z}$ influencing $\mathbf{x}$, this dependency is stored in the state $\mathbf{s}$. The formula

$$\mathcal{I}_{\text{corr}}(\mathbf{s}) \equiv \forall \mathbf{x} \in \mathit{stable} \setminus \mathit{called}.\ \sigma_\perp\,\mathbf{s}\,\mathbf{x} \sqsupseteq [\![t_{\mathbf{x}}]\!]^*(\sigma_\perp\,\mathbf{s}) \wedge \mathcal{I}_{\text{dep}}(\mathbf{x}, \mathbf{s})$$

defines the *correctness* of the state $\mathbf{s}$. This means that for every variable $\mathbf{x}$ which is solved in $\mathbf{s}$, the constraint $\sigma\,\mathbf{x} \sqsupseteq f_{\mathbf{x}}\,\sigma$ is satisfied for $\mathbf{x}$ and dependencies of $\mathbf{x}$ are treated correctly. The most difficult part of the proof was to determine invariants for the main functions of the algorithm which are sufficiently strong to prove its correctness. The most complicated invariant refers to the function $[\![\cdot]\!]'(\texttt{eval}\,\mathbf{x})$. The formula

$$\begin{aligned}
&\mathcal{I}_{[\![\cdot]\!]'(\texttt{eval}\,\mathbf{x})}(\mathbf{x}, r, \mathbf{s}, \mathbf{s}', d, \mathit{vlist}, \mathit{vlist}') \equiv \\
&\quad \mathbf{x} \in \mathit{stable} \wedge \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}) \wedge (\forall (\mathbf{y}, v) \in \mathit{vlist}.\, \mathbf{y} \in \mathit{stable}) \implies \\
&\quad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathit{vlist} \subseteq \mathit{vlist}' \wedge (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{y} \in \mathit{stable}) \wedge \\
&\quad \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\text{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}') \wedge \\
&\quad \big[\mathbf{x} \in \mathit{called} \wedge (\forall (\mathbf{y}, v) \in \mathit{vlist}.\, \mathbf{x} \in \mathit{infl}_{[\,]}\,\mathbf{y}\,\mathbf{s}) \wedge \\
&\quad \mathsf{valid}(\mathit{vlist}, \sigma_\perp\,\mathbf{s}) \wedge \mathsf{legal}(\mathit{vlist}, t_{\mathbf{x}}) \wedge \mathsf{subtree}(\mathit{vlist}, t_{\mathbf{x}}) = r \implies \\
&\qquad \big(\mathbf{x} \in \mathit{called}' \implies \\
&\qquad\quad \mathsf{valid}(\mathit{vlist}', \sigma_\perp\,\mathbf{s}') \wedge \mathsf{legal}(\mathit{vlist}', t_{\mathbf{x}}) \wedge \mathsf{subtree}(\mathit{vlist}', t_{\mathbf{x}}) = \mathsf{Answ}(d) \wedge \\
&\qquad\quad (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{x} \in \mathit{infl}_{[\,]}\,\mathbf{y}\,\mathbf{s}') \wedge \mathcal{I}_{\text{dep}}(\mathbf{x}, \mathbf{s}')\big) \wedge \\
&\qquad \big(\mathbf{x} \notin \mathit{called}' \implies \mathbf{x} \in \mathit{stable}' \setminus \mathit{called}'\big)\big]
\end{aligned}$$

relates the arguments *vlist* and $\mathbf{s}$ with the result value $((d, \mathit{vlist}'), \mathbf{s}')$ of the call whenever it terminates. It proceeds recursively on the tree r, taking as a parameter a list *vlist* of already visited variables together with their new values. The function $[\![\cdot]\!]'\,(\texttt{eval}\,\mathbf{x})$ is called for a stable variable $\mathbf{x}$ and applied to a partial path *vlist* of stable variables and an initial consistent correct state $\mathbf{s}$. As a result it returns a value d and a longer path *vlist*$'$, which extends *vlist*, of stable visited variables, together with a consistent correct state $\mathbf{s}'$. The formula states that values $\sigma\,\mathbf{x}$ of all variables $\mathbf{x}$ grew, and *infl* changes according to changes in σ. It distinguishes the case where $\mathbf{x} \in \mathit{called}$. Then if *vlist* is a valid and legal path in $t_{\mathbf{x}}$ leading to the subtree r and if $\mathbf{x} \in \mathit{called}'$ then the result path *vlist*$'$ is again valid and legal in $t_{\mathbf{x}}$ and leads to an answer d and all the dependencies of $\mathbf{x}$ are recorded. Note that by lemma 5 this implies that *vlist*$'$ is a trace in $t_{\mathbf{x}}$ by σ'. If $\mathbf{x} \in \mathit{called}$ and $\mathbf{x} \notin \mathit{called}'$ then it was reevaluated and solved during a recursive call for some variable $\mathbf{y}$ of r. It does not matter which value d is returned in this case since $\mathbf{x}$ is solved in $\mathbf{s}'$ and the corresponding constraint is satisfied and

will be preserved after the sequent update of $\sigma\,\mathbf{x}$. In the case $\mathbf{x} \notin \mathit{called}$ we can deduce that $\mathbf{x}$ is solved in $\mathbf{s}'$ using $\mathcal{I}_1(\mathbf{s}, \mathbf{s}')$. The formula

$$\begin{aligned}&\mathcal{I}_{\texttt{eval_rhs}}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l') \equiv \\ &\quad \mathbf{x} \in \mathit{called} \wedge \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}) \implies \\ &\quad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}') \wedge \\ &\quad \big[\mathbf{x} \in \mathit{called}' \implies d = [\![t_\mathbf{x}]\!]^*(\sigma_\perp\, \mathbf{s}') \wedge l' = \mathsf{trace}_\sigma\, t_\mathbf{x} \wedge \\ &\qquad (\forall (\mathbf{y}, v) \in \mathit{vlist}'.\, \mathbf{x} \in \mathit{infl}_{[]}\, \mathbf{y}\, \mathbf{s}') \wedge \mathcal{I}_{\text{dep}}(\mathbf{x}, \mathbf{s}')\big]\end{aligned}$$

relates the arguments $\mathbf{x}$ and $\mathbf{s}$ of the call of `eval_rhs x s` with the result state $\mathbf{s}'$ whenever it terminates. If the input state $\mathbf{s}$ is consistent and correct then so is the state $\mathbf{s}'$. In the case when $\mathbf{x}$ stays called we have that d is a value of the right-hand side of $\mathbf{x}$ on σ' and l' is a trace in $t_\mathbf{x}$ by σ'. In the case $\mathbf{x} \notin \mathit{called}'$ the variable $\mathbf{x}$ is processed during some intermediate recursive call and is solved in $\mathbf{s}'$. The formula

$$\begin{aligned}&\mathcal{I}_{\texttt{solve}}(\mathbf{x}, \mathbf{s}, \mathbf{s}') \equiv \\ &\quad \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}) \implies \\ &\quad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}') \wedge \\ &\quad \big[\mathbf{x} \in \mathit{stable} \implies \mathbf{s} = \mathbf{s}'\big] \wedge \\ &\quad \big[\mathbf{x} \notin \mathit{stable} \implies \mathit{stable}' \supseteq \mathit{stable} \cup \{\mathbf{x}\} \wedge \mathit{queued}' \subseteq \mathit{queued} \setminus \{\mathbf{x}\}\big]\end{aligned}$$

relates arguments $\mathbf{x}$ and $\mathbf{s}$ with the result state $\mathbf{s}'$ of the call of `solve x s` whenever it terminates. If the state $\mathbf{s}$ is consistent and correct then so is $\mathbf{s}'$. In the case $\mathbf{x} \in \mathit{stable}$ the state does not change. If $\mathbf{x} \notin \mathit{stable}$ then eventually $\mathbf{x}$ is solved in $\mathbf{s}'$ and is removed from the set *queued*. The formula

$$\begin{aligned}&\mathcal{I}_{\texttt{solve_all}}(w, \mathbf{s}, \mathbf{s}') \equiv \\ &\quad \mathcal{I}_0(\mathbf{s}) \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}) \implies \\ &\quad \mathcal{I}_0(\mathbf{s}') \wedge \mathcal{I}_1(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_\sigma(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\sigma,\mathit{infl}}(\mathbf{s}, \mathbf{s}') \wedge \mathcal{I}_{\text{corr}}(\mathbf{s}') \wedge \\ &\quad (w \cup \mathit{stable} \setminus \mathit{called} \subseteq \mathit{stable}' \setminus \mathit{called}') \wedge (\mathit{queued}' \subseteq \mathit{queued} \setminus w)\end{aligned}$$

relates the arguments w and $\mathbf{s}$ with the result state $\mathbf{s}'$ of the call `solve_all` $w\,\mathbf{s}$ whenever it terminates. It states that all the variables solved in $\mathbf{s}$ together with the variables from w are solved in $\mathbf{s}'$ and none of the variables from w is in *queued*$'$. We note that although $w = \mathit{infl}\,\mathbf{x}$ (for a corresponding $\mathbf{x}$) may contain invalid dependencies, i.e., variables not dependent on $\mathbf{x}$ on the current σ, $\mathcal{I}_{\text{corr}}(\mathbf{s}')$ states that $\mathit{infl}\,\mathbf{x}$ is appropriately recomputed.

By induction on number of unfoldings of definitions we prove in Coq that the formulae $\mathcal{I}_{\texttt{eval}}$, $\mathcal{I}_{[\![\cdot]\!]'(\texttt{eval}\,\mathbf{x})}$, $\mathcal{I}_{\texttt{eval_rhs}}$, $\mathcal{I}_{\texttt{solve}}$ and $\mathcal{I}_{\texttt{solve_all}}$ are invariants of corresponding functions in the following sense.

Theorem 6. *For all states* $\mathbf{s}, \mathbf{s}'$: `state` *the following is true:*

- *for every* $\mathbf{x}, \mathbf{y} \in V$, $d \in D$, *EvalRel*$(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{s}', d)$ *implies* $\mathcal{I}_{\texttt{eval}}(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{s}', d)$*;*
- *for every* $\mathbf{x} \in V$, $r \in \mathcal{T}(D, T)$, $d \in D$, $l, l' : (V \times D)$ *list,* *Wrap_Eval_x*$(\mathbf{x}, r, \mathbf{s}, \mathbf{s}', d, l, l')$ *implies* $\mathcal{I}_{[\![\cdot]\!]'(\texttt{eval}\,\mathbf{x})}(\mathbf{x}, r, \mathbf{s}, \mathbf{s}', d, l, l')$*;*

– *for every* $\mathbf{x} \in V$*,* $d \in D$*,* $l' : (V \times D)$ *list,* $\mathit{Eval_rhs}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l')$ *implies* $\mathcal{I}_{\mathtt{eval_rhs}}(\mathbf{x}, \mathbf{s}, \mathbf{s}', d, l')$*;*
– *for every* $\mathbf{x} \in V$*,* $\mathit{Solve}(\mathbf{x}, \mathbf{s}, \mathbf{s}')$ *implies* $\mathcal{I}_{\mathtt{solve}}(\mathbf{x}, \mathbf{s}, \mathbf{s}')$*;*
– *for every* $w \in V$ *list,* $\mathit{SolveAll}(w, \mathbf{s}, \mathbf{s}')$ *implies* $\mathcal{I}_{\mathtt{solve_all}}(w, \mathbf{s}, \mathbf{s}')$*.* □

5.4 Putting Things Together

Having verified the invariants, we now prove that theorem 4 holds, i.e., that **RLD** is a local solver. Let **s_init** be an initial state with $\mathit{stable} = \mathit{called} = \mathit{queued} = \sigma = \mathit{infl} = \emptyset$. Assume that **RLD** applied to (t, X) terminates and let $\mathbf{s}'$ be the state returned by the call `solve_all` X **s_init**. According to the definition 3, we have to show that:

1. $X \subseteq \mathit{stable}'$ and $\mathsf{dep}^*_{t,(\sigma_\bot\,\mathbf{s}')}(\mathit{stable}') \subseteq \mathit{stable}'$;
2. $\sigma_\bot\,\mathbf{s}'\,\mathbf{x} \sqsupseteq [\![t_\mathbf{x}]\!]^*(\sigma_\bot\,\mathbf{s}')$ holds for every $\mathbf{x} \in \mathit{stable}'$.

By theorem 6, implication $\mathcal{I}_{\text{solve_all}}(X, \mathbf{s_init}, \mathbf{s}')$ holds; and its premise is true, inasmuch as both $\mathcal{I}_0(\mathbf{s_init})$ and $\mathcal{I}_{\text{corr}}(\mathbf{s_init})$ hold. Therefore, we have $\mathcal{I}_1(\mathbf{s_init}, \mathbf{s}')$, and hence $\mathit{called}' = \mathit{queued}' = \emptyset$. From $(X \cup \mathit{stable} \setminus \mathit{called} \subseteq \mathit{stable}' \setminus \mathit{called}')$ we conclude, that $X \subseteq \mathit{stable}'$. From $\mathcal{I}_{\text{corr}}(\mathbf{s}')$ it follows, that $\forall \mathbf{x} \in \mathit{stable}'.\ \sigma_\bot\,\mathbf{s}'\ \mathbf{x} \sqsupseteq [\![t_\mathbf{x}]\!]^*(\sigma_\bot\,\mathbf{s}')$ and $\mathsf{dep}_{t,(\sigma_\bot\,\mathbf{s}')}(\mathit{stable}') \subseteq \mathit{stable}'$ hold. Hence we have $\mathsf{dep}^*_{t,(\sigma_\bot\,\mathbf{s}')}(\mathit{stable}') \subseteq \mathit{stable}'$ and the statement of theorem 4 follows. □

6 Conclusion

We have presented the outline of a proof that the algorithm **RLD** is a local generic solver. By that, we enabled the inclusion of this algorithm into the trusted code base of a verified program analyzer. Since the solver can be applied to constraint systems where right hand sides of variables are arbitrary *pure* functions, this enables the design and implementation of flexible and general verified analyzer frameworks.

The extended version of this paper will provide further verified properties of the algorithm **RLD**, such as sufficient conditions for its termination as well as sufficient conditions for returning fragments not of any but of the least solution of the given constraint system. In practical applications such as the analyzer Goblint it is often convenient to allow more than one constraint for a variable. Therefore, it would be also interesting to provide formalized correctness proofs also for corresponding extension of **RLD**.

References

1. Backes, M., Laud, P.: Computationally sound secrecy proofs by mechanized flow analysis. In: ACM Conference on Computer and Communications Security, pp. 370–379 (2006)
2. Cachera, D., Jensen, T.P., Pichardie, D., Rusu, V.: Extracting a data flow analyser in constructive logic. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 385–400. Springer, Heidelberg (2004)

3. Le Charlier, B., Van Hentenryck, P.: A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912 (1992)
4. Coupet-Grimal, S., Delobel, W.: A uniform and certified approach for two static analyses. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 115–137. Springer, Heidelberg (2006)
5. Fecht, C.: Gena - a tool for generating prolog analyzers from specifications. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 418–419. Springer, Heidelberg (1995)
6. Fecht, C., Seidl, H.: Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 90–104. Springer, Heidelberg (1998)
7. Fecht, C., Seidl, H.: A faster solver for general systems of equations. Sci. Comput. Program. 35(2), 137–161 (1999)
8. Hofmann, M., Karbyshev, A., Seidl, H.: What is a pure functional? In: Abramsky, S., Gavoille, C., Kirchner, C., der Heide, F.M.a., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 199–210. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14162-1_17
9. Hofmann, M., Pavlova, M.: Elimination of ghost variables in program logics. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 1–20. Springer, Heidelberg (2008)
10. Jorgensen, N.: Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 329–345. Springer, Heidelberg (1994)
11. Klein, G., Nipkow, T.: Verified bytecode verifiers. Theor. Comput. Sci. 3(298), 583–626 (2003)
12. The Coq development team. The Coq proof assistant reference manual. TypiCal Project (formerly LogiCal), Version 8.2-bugfix (2009)
13. Nipkow, T.: Verified bytecode verifiers. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 347–363. Springer, Heidelberg (2001)
14. Seidl, H., Vojdani, V.: Region analysis for race detection. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 171–187. Springer, Heidelberg (2009)
15. Seidl, H., Wilhelm, R., Hack, S.: Übersetzerbau: Analyse und Transformation. Springer, Heidelberg (2010)

Thread-Modular Counterexample-Guided Abstraction Refinement

Alexander Malkis[1], Andreas Podelski[1], and Andrey Rybalchenko[2]

[1] University of Freiburg
[2] TU München

Abstract. We consider the refinement of a static analysis method called thread-modular verification. It was an open question whether such a refinement can be done automatically. We present a counterexample-guided abstraction refinement algorithm for thread-modular verification and demonstrate its potential, both theoretically and practically.

1 Introduction

The static analysis of multi-threaded programs has been and still is an active research topic [1,3,4,5,6,8,10,11,13,14,16,17,18,27,28,29,30,35]. The fundamental problem that we address in this paper is often described by *state explosion*: the state space of a program increases exponentially in the number of its threads. As a consequence, no static analysis method scales (i.e., is polynomial) in the number of threads. While this problem has been successfully circumvented by some methods in some practical instances, we still need to investigate the principles of potential solutions ("why do they work *when* they do work").

The scheme of counterexample-guided abstraction refinement has received a lot of interest, in particular in its possible extensions from sequential to concurrent programs. Sophisticated extensions of the CEGAR scheme, as e.g. in [3, 5, 13, 14], have shown considerable success on practical examples. In this paper, we present a static analysis method based on yet another extension of the CEGAR scheme and investigate its principles. The distinguishing point of our extension lies exactly in its principles. The resulting static analysis method scales in the number of threads for a specific class of programs. As we will see, the programs in this class are rather general and many programs encountered in practice are likely to fall into this class. Thus, in many cases we are likely to obtain a principled reason for why our method succeeds on a particular example under consideration.

Another viewpoint leading to our static analysis method lies in the thread-modular proof method [11,23,27,28]. In previous work, we have shown that the proof method can be refined (manually) such that it becomes complete. Since the refinement does not work by enlarging the abstract domain (which is the basis of known refinement schemes [7, 31, 32, 33]), the questions were left open whether such a refinement can be done automatically, guided by the analysis of spurious counterexamples, and whether the resulting static analysis can be

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 356–372, 2010.
© Springer-Verlag Berlin Heidelberg 2010

global $x = y = turn = 0$

A: $x := 1$;	A: $y := 1$;
B: $turn := 1$;	B: $turn := 0$;
C: **while**(y **and** $turn$);	C: **while**(x **and not** $turn$);
critical	**critical**
D: $x := 0$; **goto** A;	D: $y := 0$; **goto** A;

Fig. 1. Peterson's mutual exclusion algorithm

efficient, in theory and/or in practice. In this paper, we give a positive answer to these questions.

The technical contribution of this paper consists of:

– An algorithm that takes the spurious counterexample produced by thread-modular abstraction and extracts the information needed for the subsequent refinement. The algorithm exploits the regularities of data structures for (unions of) Cartesian products and their operations (Cartesian products underly the abstract domain of our static analysis).
– A *thread-modular counterexample-guided abstraction refinement* that automates the fine-tuning of an existing static analysis related to the thread-modular proof method. Previously, this fine-tuning was done manually.
– A static analysis method for multi-threaded programs that scales polynomially in the number of threads for a specific class of programs. To the best of our knowledge, this is the first static analysis for which such a property is known, besides the thread-modular proof method which, however, can produce only *local proofs* (Owicki-Gries proofs without auxiliary state variables [8,11,20,27,28]).
– An implementation and an experimental evaluation indicating that the theoretical complexity guarantees can be realized in practice.

2 Illustration

In this section we provide a high level illustration of our abstraction refinement algorithm. Using the Peterson's protocol [34] for mutual exclusion and a spurious counterexample produced by thread-modular verification approach we show the refinement computed by our procedure.

See Fig. 1 for a program implementing the protocol. We want to prove the mutually exclusive access to the location labeled D, i.e., that D is not simultaneously reachable by both processes.

The thread-modular approach interleaves the computation of reachable program states for each thread with the application of Cartesian abstraction among the computed sets. For our program the reachability computation traverses the following executions of the first and second thread, respectively (where each tuple represents a valuation of the program variables x, y, $turn$, pc_1, and pc_2):

$$(0,0,0,A,A),(0,1,0,A,B),(0,1,0,A,C),(0,1,0,A,D),(1,1,0,B,D)\ ,$$
$$(0,0,0,A,A),(1,0,0,B,A),(1,0,1,C,A),(1,1,1,C,B),(1,1,0,C,C)\ .$$

The last states of the executions above, i.e., the states $(1,1,0,B,D)$ and $(1,1,0,C,C)$, have equal valuations of the global variables and hence are subject to Cartesian abstraction, which weakens the relation between the valuations of local variables of individual threads. The application of Cartesian abstraction on this pair produces the following set of states:

$$\{(1,1,0)\} \times \{B,C\} \times \{D,C\} = \{(1,1,0,B,D),(1,1,0,B,C),\\ (1,1,0,C,D),(1,1,0,C,C)\}\ .$$

The subsequent continuation of the reachability computation discovers that the first thread can reach an error state $(1,1,0,D,D)$ by making a step from the state $(1,1,0,C,D)$. That is, the thread modular approach discovers a possibly spurious counterexample to the mutual exclusion property of our program.

The feasibility of the counterexample candidate is checked by a standard backwards traversal procedure starting from the reached error state $(1,1,0,D,D)$. This check discovers that $(1,1,0,C,D)$ is the only state that can be reached backwards from $(1,1,0,D,D)$. That is, the counterexample is spurious and needs to be eliminated.

Now we apply our refinement procedure to refine the thread modular approach. First, our procedure discovers that the application of Cartesian abstraction on the pair of states $(1,1,0,B,D)$ and $(1,1,0,C,C)$ produced the state $(1,1,0,C,D)$, since

$$(1,1,0,C,D) \in \{(1,1,0)\} \times \{B,C\} \times \{D,C\}\ ,$$

and identifies it as a reason for the discovery of the spurious counterexample. Second, the Cartesian abstraction used by the thread modular approach is refined by adding $(1,1,0,B,D)$ (or, alternatively $(1,1,0,C,C)$) to the so-called *exception set* [24]. The states in the exception set are excluded from the Cartesian abstraction, thus refining it. As a result, the discovered spurious counterexample is eliminated since $(1,1,0,C,D)$ becomes unreachable. As in the existing counterexample guided abstraction refinement schemes, we proceed by applying the thread modular approach, however now it is refined by the exception set $\{(1,1,0,B,D)\}$.

In addition to the above counterexample, the thread modular approach also discovers a spurious counterexample that reaches the error state $(1,1,1,D,D)$. Our refinement procedure detects that the application of Cartesian abstraction on a state $(1,1,1,D,B)$ leads to this counterexample. Thus, the abstraction is refined by extending the exception set with the state $(1,1,1,D,B)$.

Finally, the thread modular approach refined with the resulting exception set $\{(1,1,0,B,D),(1,1,1,D,B)\}$ proves the program correct. In Section 5, we present a detailed description of how our refinement method computes exception sets.

3 Preliminaries

Now we define multithreaded programs, multithreaded Cartesian abstraction and exception sets. We combat state space explosion in the number of threads, so we keep the internal structure of a thread unspecified.

An *n-threaded program* is given by sets Glob, Loc, $\rightarrow_i$ (for $1 \leq i \leq n$), init, where each $\rightarrow_i$ is a subset of $(\text{Glob} \times \text{Loc})^2$ (for $1 \leq i \leq n$) and init $\subseteq \text{Glob} \times \text{Loc}^n$.

The components of the multithreaded program mean the following:

- The set of *shared states* Glob contains valuations of global variables
- The set of *local states* Loc contains valuations of local variables including the program counter (without loss of generality let all threads have equal sets of local states)
- $\rightarrow_i$ is the transition relation of the ith thread ($1 \leq i \leq n$).
- init is the set of initial program states.

If the *program size* $|\text{Glob}| + |\text{Loc}| + \sum_{i=1}^{n} |{\rightarrow_i}| + |\text{init}|$ is finite, the program is called *finite-state*.

The elements of $\text{States} = \text{Glob} \times \text{Loc}^n$ are called *program states*, the elements of Glob × Loc are called *thread states*.

The program is equipped with the interleaving semantics: if a thread makes a step, then it may change its own local variables and the global variables but may not change the local variables of another thread; a step of the whole program is a step of some of the threads. The *post* operator maps a set of states to the set of their successors:

$$\text{post}:\ 2^{\text{States}} \rightarrow 2^{\text{States}},$$
$$S \ \mapsto\ \{(g', l') \mid \exists\, (g, l) \in S,\, i \in \mathbb{N}_n:\ (g, l_i) \rightarrow_i (g', l'_i) \text{ and } \forall\, j \neq i:\ l_j = l'_j\},$$

where $\mathbb{N}_n$ is the set of first n positive integers and the lower indices denote components of a vector. The verification goal is to show that any computation that starts in an initial state stays within the set of safe states, formally:

$$\bigcup_{k \geq 0} \text{post}^k(\text{init}) \subseteq \text{safe}\,.$$

Thread-modular reasoning can prove the same properties as abstract fixpoint checking in the following setup [23,22]:

$D = \mathfrak{P}(\text{States})$ is the concrete domain, ordered by inclusion,
$D^{\#} = (\mathfrak{P}(\text{Glob} \times \text{Loc}))^n$ is the abstract domain, least upper bound $\sqcup$ is the componentwise union,

$$\begin{array}{llll} \alpha_{\text{mc}}: D & \rightarrow D^{\#}, & S \mapsto (\{(g, l_i) \mid (g, l) \in S\})_{i=1}^{n}, \\ \gamma_{\text{mc}}: D^{\#} & \rightarrow D, & T \mapsto \{(g, l) \mid \forall\, i \in \mathbb{N}_n : (g, l_i) \in T_i\}, \end{array}$$

are the abstraction and concretization maps which form the multithreaded Cartesian Galois connection. Interestingly, the Owicki-Gries proof method without auxiliary variables [27] can prove exactly the same properties [20].

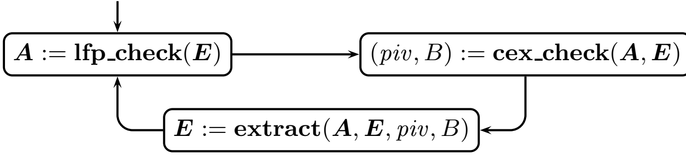


Fig. 2. TM-CEGAR: topmost level. The function **lfp_check** tries to compute an inductive invariant by generating the sequence $\boldsymbol{A}$ by abstract fixpoint iteration where $\boldsymbol{E}$ tunes the interpretation of elements of $\boldsymbol{A}$. In case an error state occurs in $\boldsymbol{A}$, the function **cex_check** determines the reason for the error occurrence. It determines the smallest iterate index *piv* such that the interpretation of A_{piv} has erroneous descendants later in $\boldsymbol{A}$. The function **extract** looks at the way A_{piv} was constructed, at those states in the concretization of this iterate that have erroneous successors, and tunes the parameters starting from E_{piv}.

Given a set of states $E \subseteq \text{States}$, the *exceptional* Galois connection

$$\begin{aligned} \alpha_E &: D \to D, \quad S \mapsto S \setminus E\,, \\ \gamma_E &: D \to D, \quad S \mapsto S \cup E\,. \end{aligned}$$

can be used to parameterize any abstract interpretation. In particular, the *parameterized multithreaded Cartesian* Galois connection

$$(\alpha_{\text{mc},E},\ \gamma_{\text{mc},E}) \quad = \quad (\alpha_{\text{mc}} \circ \alpha_E,\ \gamma_E \circ \gamma_{\text{mc}})$$

allows arbitrary precise polynomial-time analysis by a clever choice of the *exception set* E [24].

How to find a suitable exception set in acceptable time automatically? The remainder of the article deals with this question.

4 Algorithm

Now we show TM-CEGAR, a thread-modular counterexample-guided abstraction refinement loop, that, given a multithreaded program and a set of error states, proves or refutes nonreachability of error states from the initial ones.

The computation of TM-CEGAR on a particular multithreaded program is a sequence of *refinement phases*, such that within each refinement phase, previously derived exception sets are used for the fixpoint iteration and a new exception set is computed. A refinement phase corresponds to one execution of the CEGAR loop.

TM-CEGAR operates on two sequences $\boldsymbol{A} = (A_i)_{i\geq 1} \in D^{\#^\omega}$ and $\boldsymbol{E} = (E_i)_{i\geq 1} \in D^\omega$. The sequence $\boldsymbol{A}$ is filled by the iterates of the abstract fixpoint computation, where each iterate A_i has a different interpretation which depends on E_i $(i \geq 1)$.

The topmost level of TM-CEGAR is given in Fig. 2 (variables printed in bold face are sequences). Initially, the sequence of parameters $\boldsymbol{E}$ consists of empty

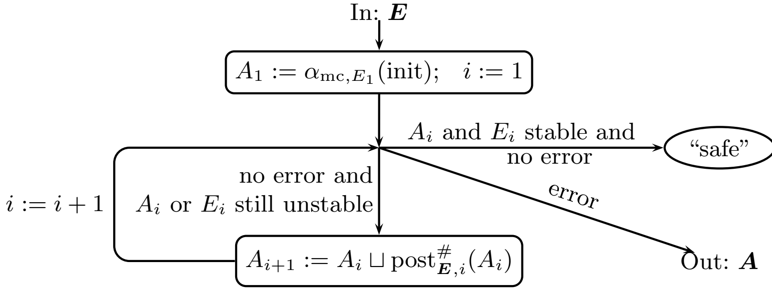


Fig. 3. The **lfp_check** function. The function $\mathrm{post}^{\#}_{\boldsymbol{E},i}$ is an abstract transformer whose precision is tuned by particular elements from $\boldsymbol{E}$. An error state is detected when $\gamma_{\mathrm{mc},E_i}(A_i) \not\subseteq \mathrm{safe}$. Stability of A_i and E_i means that $(E_{i-1}, A_{i-1}) = (E_i, A_i)$.

sets. Let's fix a refinement phase, assuming that the sequence of parameters has already been constructed previously. Using parameters from $\boldsymbol{E}$, we construct the sequence of iterates $\boldsymbol{A}$ in the function **lfp_check**. Assuming abstract fixpoint computation has found error states in some iterate of $\boldsymbol{A}$, the function **cex_check** examines $\boldsymbol{A}$ to find the earliest states in $\boldsymbol{A}$ that are ancestors of the found error states. In case these earliest states don't contain initial ones, but lie in the interpretation of some iterate A_{piv}, the interpretations of A_{piv} and of all subsequent iterates are tuned by changing the parameters from E_{piv} onwards.

4.1 Abstract Reachability Analysis

The abstract fixpoint computation in Fig. 3 generates the sequence of iterates $\boldsymbol{A}$ based on the sequence of parameters $\boldsymbol{E}$.

The **lfp_check** function generates the first element of $\boldsymbol{A}$ as an abstraction of the initial states, parameterized by the first parameter: $A_1 = \alpha_{\mathrm{mc},E_1}(\mathrm{init})$. The subsequent iterates are computed by taking the join of the current iterate with an approximation of post, applied to the current iterate. The approximation of post is tuned by $\boldsymbol{E}$:

$$\mathrm{post}^{\#}_{\boldsymbol{E},i} = \alpha_{\mathrm{mc},E_{i+1}} \circ \mathrm{post} \circ \gamma_{\mathrm{mc},E_i} .$$

The computation of iterates stops in two cases:

– Either the concretizations of the iterates remain in the safe states and no more grow, which happens when the sequences $\boldsymbol{A}$ and $\boldsymbol{E}$ get stable after some index;
– Or the concretization of some iterate contains an error state.

In the first case, TM-CEGAR has proven correctness of the program and thus exits immediately.

In the second case both sequences $\boldsymbol{A}$ and $\boldsymbol{E}$ are analyzed.

To optimize **lfp_check**, notice that the new and the old sequences of parameters share a common prefix (empty prefix in the worst case): say, $E_1 \dots E_j$

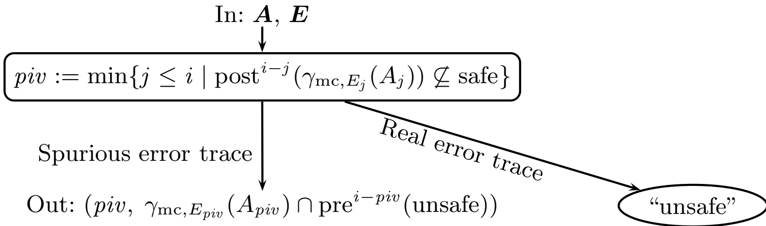


Fig. 4. The high-level view of the **cex_check** function. The set unsafe is States $\setminus$ safe. A real error trace is detected when $piv = 1 \wedge \mathrm{post}^{i-1}(\mathrm{init}) \not\subseteq \mathrm{safe}$. A spurious error is detected when $(piv = 1 \wedge \mathrm{post}^{i-1}(\mathrm{init}) \subseteq \mathrm{safe}) \ \vee \ piv > 1$.

remained the same for some $j \geq 1$. Then $A_1 \dots A_j$ remain the same and don't have to be recomputed in the next refinement phase. This optimization doesn't have any influence on the asymptotic runtime, but is a great help in practice.

4.2 Checking Counterexample for Spuriousness

The **cex_check** function assumes that error states are found in concretization of the iterate A_i and determines the earliest ancestors of those error states in $\boldsymbol{A}$.

To implement the high-level description of **cex_check** in Fig. 4, we compute the precise ancestors of the error states inside the concretizations of the iterates backwards. For that, we construct *bad regions* $\mathrm{Bad}_{piv}, \dots, \mathrm{Bad}_i$ as follows:

$$\mathrm{Bad}_i := \gamma_{\mathrm{mc},E_i}(A_i) \setminus \mathrm{safe}\,,$$
$$\mathrm{Bad}_{j-1} := \mathrm{pre}(\mathrm{Bad}_j) \cap \gamma_{\mathrm{mc},E_{j-1}}(A_{j-1}) \text{ for } j \leq i$$

until the bad region gets empty. The smallest iterate number piv for which the bad region is nonempty is called *pivot*. If pivot is 1 and there are initial states in Bad_1, the program has an error trace. Otherwise the error detection was due to the coarseness of the abstraction; another abstraction has to be chosen for the pivot iterate and subsequent iterates.

4.3 Refine: Extract New Exception Set

Once a pivot iterate number piv is found, the exception set E_{piv} has to be enlarged to exclude more states from approximation. It is not obvious how to do that. We first specify requirements to the **extract** function, and then show the best known way of satisfying those requirements. Implementation variants of **extract** are discussed afterwards.

Requirements to extract. The precondition of **extract** is that $\emptyset \neq \mathrm{Bad}_{piv} \subseteq \gamma_{\mathrm{mc},E_{piv}}(A_{piv})$, and

- neither the interpretation of the previous iterate, namely, $\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1})$,
- nor the successors of that interpretation

intersect Bad_{piv}. (Otherwise forward search would hit error states one iterate earlier or Bad_{piv-1} were nonempty.)

The postconditions imposed on the output of $\widetilde{\boldsymbol{E}}$ of **extract** are:

- $\gamma_{\mathrm{mc},\widetilde{E_{piv}}} \circ \alpha_{\mathrm{mc},\widetilde{E_{piv}}} \circ \mathrm{post} \circ \gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1})$ doesn't intersect Bad_{piv} and
- $E_{piv} \subseteq \widetilde{E_{piv}} \subseteq E_{piv} \cup \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ and
- $\widetilde{E_k} = E_k$ for $k < piv$ and
- $\widetilde{E_k} = E_{piv}$ for $k > piv$.

The first postcondition ensures that no error trace starting at position piv and ending at position i would occur in **lfp_check** in the next refinement round. The second postcondition makes certain that previous spurious counterexamples starting at the pivot position would not reappear and that no new overapproximation is introduced. The third postcondition provides sufficient backtracking information for the next refinement phases. The last postcondition saves future computation time, intuitively, conveying the already derived knowledge to the future refinement phases; it may be relaxed, as we will see when discussing algorithm variants. The postconditions establish a monotonously growing sequence of parameters, and guarantee that the next sequence of interpretations of iterates is lexicographically smaller than the previous one, ensuring progress.

Implementation of extract. We gradually reduce the extraction problem to simpler subproblems.

First, we choose a set $\Delta E \subseteq \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ such that $\gamma_{\mathrm{mc},\Delta E} \circ \alpha_{\mathrm{mc},\Delta E} \circ \mathrm{post} \circ \gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1})$ doesn't intersect Bad_{piv}. Then we let $\widetilde{E_k} = E_k$ for $k < piv$ and $\widetilde{E_k} = \Delta E \cup E_{piv}$ for $k \geq piv$.

To choose such ΔE, we divide A_{piv-1}, $\mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))$ and Bad_{piv} into smaller elements of the abstract and concrete domains, such that the shared state within each small element is constant $g \in \mathrm{Glob}$:

$$\begin{aligned} A^{(g)} &= (\{(g,l) \in (A_{piv-1})_i\})_{i=1}^n \,, \\ P^{(g)} &= \{(g,l) \in \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1}))\} \,, \\ B^{(g)} &= \{(g,l) \in \mathrm{Bad}_{piv}\} \,. \end{aligned}$$

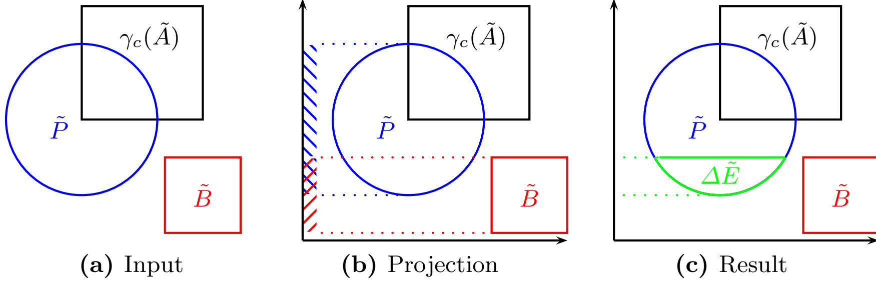
Then

$$\begin{aligned} A_{piv-1} &= \textstyle\bigsqcup_{g\in\mathrm{Glob}} A^{(g)}_{piv-1} \,, \\ \mathrm{post}(\gamma_{\mathrm{mc},E_{piv-1}}(A_{piv-1})) &= \textstyle\bigcup_{g\in\mathrm{Glob}} P^{(g)} \,, \\ \mathrm{Bad}_{piv} &= \textstyle\bigcup_{g\in\mathrm{Glob}} B^{(g)} \,. \end{aligned}$$

For each $g \in \mathrm{Glob}$, we have to find an exception set $\Delta^{(g)} \subseteq P^{(g)}$ such that $\gamma_{\mathrm{mc},\Delta E^{(g)}} \circ \alpha_{\mathrm{mc},\Delta E^{(g)}}(P^{(g)})$ doesn't intersect $B^{(g)}$. After having found such sets, we let $\Delta E = \bigcup_{g\in\mathrm{Glob}} \Delta E^{(g)}$.

Assume we have fixed $g \in \mathrm{Glob}$ and want to find $\Delta^{(g)}$ as above. To do that, it suffices to solve a similar problem for the standard Cartesian abstraction:

$$\begin{aligned} \alpha_c &: \ \mathfrak{P}(\mathrm{Loc}^n) \to (\mathfrak{P}(\mathrm{Loc}))^n \,, && S \mapsto (\pi_i(S))_{i=1}^n \,, \\ \gamma_c &: \ (\mathfrak{P}(\mathrm{Loc}))^n \to \mathfrak{P}(\mathrm{Loc}^n) \,, && ((T_i)_{i=1}^n) \mapsto \textstyle\prod_{i=1}^n T_i \,, \end{aligned}$$



(a) Input **(b)** Projection **(c)** Result

Fig. 5. Internals of **extract**$(\boldsymbol{A}, \boldsymbol{E}, piv, B)$

where π_i projects a set of tuples to the ith component and index c means Cartesian. Namely, we are given a tuple $\tilde{A} \in (\mathfrak{P}(\text{Loc}))^n$ and sets $\tilde{P}, \tilde{B} \subseteq \text{Loc}^n$ such that $\tilde{B} \cap (\gamma_c(\tilde{A}) \cup \tilde{P}) = \emptyset$, and we want to find $\Delta\tilde{E} \subseteq \tilde{P}$ such that $\tilde{B} \cap \gamma_c \circ \alpha_c(\gamma_c(\tilde{A}) \cup (\tilde{P} \setminus \Delta\tilde{E})) = \emptyset$.

To solve this problem, we take the representation of $\tilde{B}$ as a union of products, say, $\tilde{B} = \bigcup_{j=1}^{m} \tilde{B}^{(j)}$ where $\tilde{B}^{(j)} = \prod_{i=1}^{n} \tilde{B}_i^{(j)}$ $(1 \leq j \leq m)$. Then we solve the problem for each $B^{(j)}$ instead of $\tilde{B}$ separately, and then take the union of the results.

So now let $j \in \mathbb{N}_m$ be fixed and let $\tilde{B} = \prod_{i=1}^{n} \tilde{B}_i$ be a Cartesian product such that $\tilde{B} \cap (\gamma_c(\tilde{A}) \cup \tilde{P}) = \emptyset$, as depicted on an example in Fig. 5a. We want to find $\Delta\tilde{E} \subseteq \tilde{P}$ such that $\tilde{B} \cap \gamma_c \circ \alpha_c(\gamma_c(\tilde{A}) \cup (\tilde{P} \setminus \Delta\tilde{E})) = \emptyset$. Since $\tilde{B}$ and $\gamma_c(\tilde{A})$ are products that don't intersect, there is a dimension $i \in \mathbb{N}_n$ such that $\tilde{B}_i$ and $\tilde{A}_i$ are disjoint (where $\tilde{A} = (\tilde{A}_i)_{i=1}^{n}$). In example on Fig. 5b, this is the vertical dimension $i = 2$. We let $\Delta\tilde{E} = \{p \in \tilde{P} \mid p_i \in B_i\}$, as in Fig. 5c.

Notice that $\tilde{P} \setminus \Delta\tilde{E}$ has no points whose i^{th} component occurs as the ith component of a point of $\tilde{B}$. Thus the projections of two sets $\tilde{B}$ and of $\gamma_c \circ \alpha_c(\gamma_c(\tilde{A}) \cup \tilde{P} \setminus \Delta\tilde{E})$ onto the ith component are disjoint. Thus the two sets are disjoint.

Variants of extract. Now we discuss another way of satisfying the stated postcondition of **extract** as well as a variant of those postconditions.

It turns out that taking not just one dimension in which $\tilde{B}_i$ and $\tilde{A}_i$ are disjoint, but all such dimensions (and solving the problem for each of the dimensions, and taking the union of the results), creates slightly larger sets on many examples, but saves future refinement phases in general. We call this variant of **extract** the *eager* variant. The total runtime is decreased by a factor between 1 and 2, so we optionally use this variant in practice.

We may avoid more future refinement steps by creating exception sets not only for the iterate number *piv*, but also for as many iterate numbers between *piv* and i as possible, using, e.g., Bad_{piv} till Bad_i for B. This optimization requires a relaxed postcondition of **extract**. However, the effect of this optimization was insignificant on all the examples.

5 Applying TM-CEGAR to Peterson's Protocol

In Section 2 we have sketched the main steps of TM-CEGAR on Peterson's protocol. Now we show the computation in more detail.

In the initial refinement phase, the sequence of exception sets $\boldsymbol{E}$ contains empty sets only. The procedure **lfp_check** starts with the iterate

$$A_1 \;=\; (\{(0,0,0,A)\},\; \{(0,0,0,A)\})\,,$$

where each tuple represents a valuation of program variables x, y, $turn$, pc. The **lfp_check** computation arrives at iterates (we skip A_2, A_3 as well as uninteresting states not having shared parts $(1,1,0)$ or $(1,1,1)$)

$$\begin{aligned}
A_4 \;=\; & (\{(1,1,0)\} \times \{B\} \cup \{(1,1,1)\} \times \{C\} \cup \ldots, \\
& \{(1,1,0)\} \times \{B, C\} \cup \{(1,1,1)\} \times \{B\} \cup \ldots), \\
A_5 \;=\; & (\{(1,1,0)\} \times \{B, C\} \cup \{(1,1,1)\} \times \{C, D\} \cup \ldots, \\
& \{(1,1,0)\} \times \{B, C, D\} \cup \{(1,1,1)\} \times \{B, C\} \cup \ldots), \\
A_6 \;=\; & (\{(1,1,0)\} \times \{B, C, D\} \cup \{(1,1,1)\} \times \{C, D\} \cup \ldots, \\
& \{(1,1,0)\} \times \{B, C, D\} \cup \{(1,1,1)\} \cup \{B, C, D\} \cup \ldots).
\end{aligned}$$

The iterate A_6 is the earliest one whose concretization $\gamma_{\mathrm{mc},E_6}(A_6)$ contains error states, in this case $(1,1,0,D,D)$ and $(1,1,1,D,D)$. The forward computation detects those error states and hands $\boldsymbol{A}$ over to **cex_check**.

Notice that possible predecessors of the detected error states, namely, $(1,1,0,C,D)$ and $(1,1,1,D,C)$, are in the concretization of A_5. However, those states have no predecessors at all, thus the pivot iterate is 5. So **cex_check** returns $piv = 5$ and $B = \{(1,1,0,C,D),(1,1,1,D,C)\}$ and hands those values over to **extract**.

Procedure **extract** considers shared states $(1,1,0)$ and $(1,1,1)$ separately.

For shared state $(1,1,0)$, **cex_check** is given the tuple $\tilde{A} = (\{B\},\{B,C\})$ (obtained from A_4) and sets $\tilde{P} = \{B\} \times \{B,C,D\} \cup \{(C,C)\}$ (obtained from the successors of the concretization of A_4), $\tilde{B} = \{(C,D)\}$ (obtained from B). Notice that $\tilde{B}$ consists of one point only, which is trivially a product. Since $\tilde{A}_2 \cap \pi_2(\tilde{B}) = \{B,C\} \cap \{D\} = \emptyset$, **extract** can choose $\Delta\tilde{E} = \{p \in \tilde{P} \mid p_2 \in \pi_2(\tilde{B})\} = \{(B,D)\}$.

For shared state $(1,1,1)$, **extract** chooses $\Delta\tilde{E} = \{(D,B)\}$ analogously.

Thus the generated exception set is $\{(1,1,0,B,D),(1,1,1,D,B)\}$, which **extract** assigns to E_5, E_6, E_7, E_8, ... The exceptions sets before the pivot, namely, E_1 till E_4, remain empty.

The next forward computation proceeds as the previous one till and including the iterate 4, and the iterate 5 is smaller than the previous one:

$$\begin{aligned}
A_5 = & (\{(1,1,0)\} \times \{B, C\} \cup \{(1,1,1)\} \times \{C\} \cup \ldots, \\
& \{(1,1,0)\} \times \{B, C\} \cup \{(1,1,1)\} \times \{B, C\} \cup \ldots).
\end{aligned}$$

The abstract fixpoint computation terminates at iterate 8 without finding an error state:

$$\begin{aligned} A_8 =(&\{(0,0,0,A),(0,0,1,A),(0,1,0,A),(0,1,1,A),(1,0,0,B)\} \\ &\cup \{(1,0,1),(1,1,0)\} \times \{B,C,D\} \cup \{(1,1,1)\} \times \{B,C\}, \\ &\{(0,0,0,A),(0,0,1,A)\} \cup \{(0,1,0)\} \times \{B,C,D\} \cup \{(0,1,1,B),(1,0,0,A)\} \\ &\cup \{(1,0,1,A)\} \cup \{(1,1,0)\} \times \{B,C\} \cup \{(1,1,1)\} \times \{B,C,D\}) \end{aligned}$$

Its concretization $\gamma_{\text{mc},E_8}(A_8)$ is an inductive invariant that contains no state of the form $(_,_,_,D,D)$, so mutual exclusion is proven.

6 Parallel Mutex Loop

Now we will describe a practically interesting infinite class of programs. We show that TM-CEGAR can verify the programs of the class in polynomial time. We also show that our implementation can cope with the class better than the state-of-the-art tool SPIN.

The most basic synchronization primitive, namely, a binary lock, is widely used in multithreaded programming. For example, Mueller in Fig. 2 in [25] presents a C function that uses binary locks from the standard Pthreads library [2,19], through calls to `pthread_mutex_lock` and `pthread_mutex_unlock`. The first function waits until the lock gets free and acquires it in the same transition, the second function releases the lock. Since we care about the state explosion problem in the number of threads only, we abstract away the shared data and replace the local operations by skip statements which change control flow location only.

Our class is defined by programs in Fig. 6. In a program of the class, each of n threads executes a big loop (a variant of the class in which threads have no loops but are otherwise the same has the same nice properties), inside of a loop a lock is acquired and released m times, allowing $k-1$ local operations inside each critical section. E.g. for the example of Mueller we have $k=3$, $m=1$, an unspecified n. This class extends the class presented in [24] by allowing variably long critical sections.

The property to be proven is mutual exclusion: no execution should end in a state in which some two threads are in their critical sections.

For a human, it might seem trivial that mutual exclusion holds. However, given just the transition relation of the threads, verification algorithms do have problems with the programs of the class. In fact, Flanagan and Qadeer [11] have shown a very simple program of this class that cannot be proven by thread-modular verification. Actually, it can be shown that no program of the class has a thread-modular proof.

6.1 Polynomial Runtime

Now we will show that our algorithm proves the correctness of mutex programs in polynomial time.

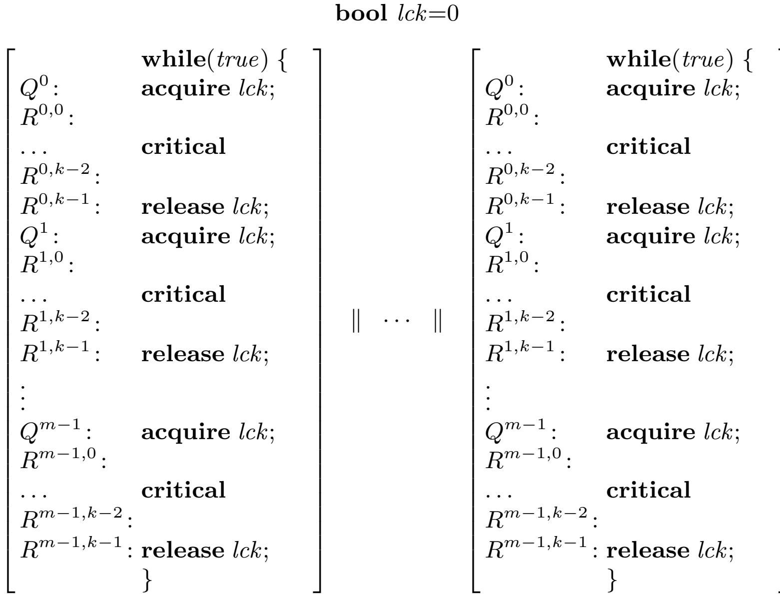


Fig. 6. Schema for programs consisting of n concurrent threads with m critical sections per thread such that each critical section has k control locations. The statement "**acquire** *lck*" waits until $lck = 0$ and sets it to 1. The statement "**release** *lck*" sets *lck* to 0. This class admits polynomial, efficient, precise and automatic thread-modular verification.

Theorem 1. *The runtime of TM-CEGAR on a program from the mutex class is polynomial in the number of threads n, number of critical sections m and size of the critical section k.*

Proof. Let $C = \{R^{j,l} \mid j < m \text{ and } l < k\}$ be the critical local states, $N = \{Q^j \mid j < m\}$ the noncritical local states and $\text{Loc} = C \,\dot\cup\, N$ the local states of a thread. A state (g, l) is an error state iff

$$\exists\, i, j \in \mathbb{N}_n : i \neq j \text{ and } a_i \in C \text{ and } a_j \in C\,.$$

For the proof of polynomial complexity we choose an eager version of **extract**, which is simpler to present. The eager version creates symmetrical exception sets for symmetrical inputs of **extract**: if interchanging two threads doesn't change the input of **extract**, the output is also stable under thread swapping.

The CEGAR algorithm needs mk refinement phases. In each phase, a new critical location is discovered. More specifically, in phases jk ($j < m$), the locations Q^j and $R^{j,0}$ are discovered. In phases $jk + r$ ($j < m$ and $1 \leq r < k$), the location $R^{j,r}$ is discovered. In each phase at least one new location is discovered because the set of error states has no predecessors and backtracking is not necessary. At the same time, no more than one critical location per phase

is discovered: due to symmetry, when a new critical location of one thread is discovered, so it happens for all the threads. Since this critical location is new, it is not in the current exception set, thus it gets subjected to Cartesian abstraction, which leads to tuples with n critical locations (because of symmetry). Then the error states are hit and the exception set is enlarged. The eager version of **extract** produces, simplifying, all tuples where one component is critical and might include the new location (and the critical locations of the previous critical sections) and the other components are noncritical. This new exception set turns out to be equal to the current set of successors in their critical sections (if it were not, the difference between the successors and the exception set had at least one critical location, and, by symmetry, at least n, which would lead to error states after approximation). Subtracting the exception set from the successor set produces only tuples of noncritical locations, which get abstracted to a product of noncritical locations.

We just provide the central computation result, namely the exception set for each phase $jk + r$ $((j < m \text{ and } r < k) \text{ or } (j = m \text{ and } r = 0))$; for details on intermediate exception sets, see [21]. We are interested in asymptotic behavior and thus show the derived exception set for large parameter values $n \geq 3$, $m \geq 1$ and $k \geq 2$ (for smaller values the exception sets are simpler).

Let $B(U, V)$ be the union over n-dimensional products in which exactly exactly one component set is V and the remaining are U. Now
$E_1 = \emptyset$ and
$E_{p(k+1)+2+l} = \{1\} \times (B(\{Q^{p'} \mid p' < p\}, \{R^{p',l'} \mid p' < p \wedge l' < k\}) \cup B(\{Q^{p'} \mid p' \leq p\}, \{R^{p',l'} \mid p' \leq p \wedge l' \leq \min\{l, k-1\}\}))$, whose maximized form is
$\{1\} \times (B(\{Q^{p'} \mid p' < p\}, \{R^{p',l'} \mid (p' < p \wedge l' < k) \vee (p' \leq p \wedge l' \leq \min\{l, k-1\})\}) \cup B(\{Q^{p'} \mid p' \leq p\}, \{R^{p',l'} \mid p' \leq p \wedge l' \leq \min\{l, k-1\}\}))$ for $p < j$ and $l \leq k$ as well as for $p = j$ and $l < r$,
the ultimate exception set is $E_{j(k+1)+1+r}$.

This representation is maximized: if some product is a subset of restriction of $E_{p(k+1)+2+l}$ to shared part 0 (resp. to shared part 1), it is a subset of some of the products in the above representation for shared part 0 (resp. for shared part 1).

Since each exception set has a polynomial-size maximized form, each refinement phase is polynomial-time by [24]. The number of refinement phases is also polynomial, so the total runtime is polynomial. □

6.2 Experiments

We have implemented TM-CEGAR in OCAML and ran tests on a 3MHz Intel machine.

We compared TM-CEGAR to the existing state-of-the-art tool SPIN 5.2.4 [15]. For comparison, we have fixed $k = 1$ locations per critical section and $m = 3$ critical sections per thread, then we measured the runtimes of TM-CEGAR and SPIN in dependency on the number of threads n. We encoded the mutual exclusion property for SPIN by a variable which is incremented on

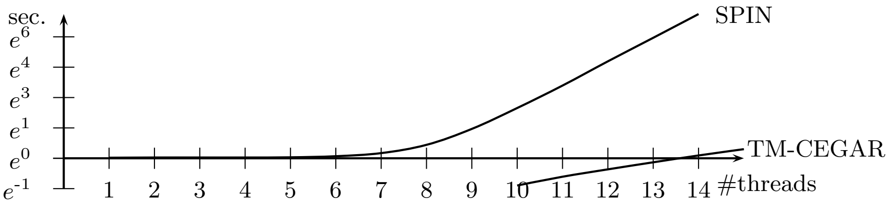


Fig. 7. SPIN vs. TM-CEGAR for 3 critical sections with one location each. Here $e \approx 2.7$ is the basis of natural logarithms. SPIN ran in exponential time $O(3.2^n)$, requiring 1892 seconds for 14 threads. TM-CEGAR needed only polynomial time $O(n^5)$, requiring around a second for 14 threads.

$m \setminus n$	1	10	20	30	40	50	60	70
1	0	0.30	6.94	46.72	185.51	553.47	1363.07	2912.25
3	0	10.90	235.36	1571.93	6085.92	17778.27	42848.10	90483.99
5	0	35.64	790.79	5260.06	20744.28	60610.99	147084.53	310593.29
7	0	80.29	1820.60	12276.20	48708.02	142755.15	346740.45	736117.10
9	0.01	150.52	3455.05	23538.67	94063,06	276296.15	671723.67	1432164.26

Fig. 8. Runtimes on the locks class for critical sections of size $k = 9$, a variable number of threads n and a variable number of critical sections m

acquires and decremented on releases, the property to be checked is that the value of this variable never exceeds one. The runtimes of SPIN and TM-CEGAR are depicted in Fig. 7 on a logarithmic scale.

SPIN fails at 15 threads, exceeding the 1 GB space bound, even if the most space-conserving switches are used (if default switches are used, SPIN runs out of space for 12 threads already after 12 seconds). Compared to that, TM-CEGAR has a tiny runtime, requiring around a second for 14 threads.

Fig. 8 demonstrates the behavior of TM-CEGAR on large examples. Of course, it is infeasible to wait for the completion of the algorithm on very large instances in practice. But we were astonished to see that TM-CEGAR requires negligibly small space. For example, after running for 3.4 days on the instance $n = 100$ threads, $m = 9$ critical sections of size $k = 1$, TM-CEGAR consumed at most 100MB; while after running for half a month on the instance $n = 80$, $k = m = 7$, it consumed only 150MB.

7 Related Work

Our work builds upon the thread-modular analysis to the verification of concurrent programs [11], which is based on an adaptation on the Owicki-Gries proof method [28] to finite state systems.

In this paper we address the question of improving the precision of thread modular analysis automatically, thus overcoming the inherent limitation of [11] to local proofs. We automate our previous work on exception sets [24] (which requires user interaction) by exploiting spurious counterexamples.

An alternative approach to improve the precision of thread modular analysis introduces additional global variables that keep track of relations between valuations of local variables of individual threads [5]. As in our case, this approach is guided by spurious counterexamples. In contrast, our approach admits a complexity result on a specific class of programs for which the analysis is polynomial in the number of threads. Identifying a similar result for the technique in [5] is an open problem.

Keeping track of particular correlations between threads can be dually seen as losing information about particular threads [9,12]. Formally connecting CEGAR-TM with locality-based abstractions as well as the complexity analysis for the latter is an open problem.

Extensions for dealing with infinite-state systems (in rely-guarantee fashion) are based on counterexample-guided schemes for data abstraction [13]. While our method takes a finite-state program as input, we believe it can be combined with predicate abstraction over data to deal with infinite-state systems.

8 Conclusion

In this paper, we have presented the following contributions.

- An algorithm that takes the spurious counterexample produced by thread-modular abstraction and extracts the information needed for the subsequent refinement. The algorithm exploits the regularities of data structures for (unions of) Cartesian products and their operations.
- A *thread-modular counterexample-guided abstraction refinement* that automates the fine-tuning of an existing static analysis related to the thread-modular proof method. Previously, this fine-tuning was done manually.
- A static analysis method for multi-threaded programs that scales polynomially in the number of threads, for a specific class of programs. To the best of our knowledge, this is the first static analysis for which such a property is known, besides the thread-modular proof method which, however, can produce only *local proofs*.
- An implementation and an experimental evaluation indicating that the theoretical complexity guarantees can be realized in practice.

So far, we have concentrated on the state-explosion problem for multi-threaded programs, the concrete algorithms assume a finite-state program as an input. The assumption is justified if, e.g., one abstracts each thread. Doing so in a preliminary step may be too naive. Thus, an interesting topic for future work is the interleaving of thread-modular abstraction refinement with other abstraction refinement methods, here possibly building on the work of, e.g., [13,14].

References

1. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551 (2003)
2. Bradford Nichols, J.P.F., Buttlar, D.: Pthreads programming. O'Reilly & Associates, Inc, Sebastopol (1996)
3. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
4. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
5. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 55–67. Springer, Heidelberg (2007)
6. Cousot, P., Cousot, R.: Invariance proof methods andanalysis techniques for parallel programs. In: Automatic Program Construction Techniques, pp. 243–271. Macmillan, Basingstoke (1984)
7. Cousot, P., Ganty, P., Raskin, J.-F.: Fixpoint-guided abstraction refinements. In: Nielson and Filé [26], pp. 333–348
8. de Roever, W.-P.: A compositional approach to concurrency and its applications. Manuscript (2003)
9. Esparza, J., Ganty, P., Schwoon, S.: Locality-based abstractions. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 118–134. Springer, Heidelberg (2005)
10. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. 338(1-3), 153–183 (2005)
11. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
12. Ganty, P.: The Fixpoint Checking Problem: An Abstraction Renement Perspective. PhD thesis, Université Libre de Bruxelles (2007)
13. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: Pugh, W., Chambers, C. (eds.) PLDI, pp. 1–13. ACM, New York (2004)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
15. Holzmann, G.: The Spin model checker: Primer and reference manual. Addison-Wesley, Reading ISBN 0-321-22862-6, http://www.spinroot.com
16. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)
17. Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic reduction of thread interleavings in concurrent programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS. LNCS, vol. 5505, pp. 124–138. Springer, Heidelberg (2009)
18. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
19. Leroy, X.: Pthreads linux manual pages, http://www.digipedia.pl/man/pthread_mutex_init.3thr.html

20. Malkis, A.: Cartesian Abstraction and Verification of Multithreaded Programs. PhD thesis, Albert-Ludwigs-Universität Freiburg (2010)
21. Malkis, A., Podelski, A.: Refinement with exceptions. Technical report (2008), http://www.informatik.uni-freiburg.de/ alexmalk/refinementWithExceptions_techrep.pdf
22. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular verification and Cartesian abstraction. In: Presentation at TV 2006 (2006)
23. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular verification is Cartesian abstract interpretation. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 183–197. Springer, Heidelberg (2006)
24. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: Nielson and Filé [26], pp. 218–232
25. Mueller, F.: Implementing POSIX threads under UNIX: Description of work in progress. In: Proceedings of the 2nd Software Engineering Research Forum, Melbourne, Florida (November 1992)
26. Nielson, H.R., Filé, G. (eds.): SAS 2007. LNCS, vol. 4634. Springer, Heidelberg (2007)
27. Owicki, S.S.: Axiomatic Proof Techniques For Parallel Programs. PhD thesis, Cornell University, Department of Computer Science, TR 75-251 (July 1975)
28. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. 6, 319–340 (1976)
29. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
30. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI 2004, pp. 14–24. ACM, New York (2004)
31. Giacobazzi, F.S.R., Ranzato, F.: Making abstract interpretations complete. JACM (2000)
32. Ranzato, F., Rossi-Doria, O., Tapparo, F.: A forward-backward abstraction refinement algorithm. In: Logozzo, F., Peled, D., Zuck, L. D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 248–262. Springer, Heidelberg (2008)
33. Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. Log. Comput. 17(1), 157–197 (2007)
34. Shankar, A.U.: Peterson's mutual exclusion algorithm (2003), http://www.cs.umd.edu/~shankar/712-S03/mutex-peterson.ps
35. Vineet Kahlon, F.I., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)

Generating Invariants for Non-linear Hybrid Systems by Linear Algebraic Methods

Nadir Matringe[2], Arnaldo Vieira Moura[3], and Rachid Rebiha[1,3,⋆]

[1] Faculty of Informatics, University of Lugano, Switzerland
rachid.rebiha@lu.unisi.ch

[2] Institue de Mathematiques de Jussieu Université Paris 7-Denis Diderot, France
nadir.matringe@univ-jussieu.fr

[3] Institute of Computing, University of Campinas, SP.Brasil
arnaldo@ic.unicamp.br

Abstract. We describe powerful computational methods, relying on linear algebraic methods, for generating ideals for non-linear invariants of algebraic hybrid systems. We show that the preconditions for discrete transitions and the Lie-derivatives for continuous evolution can be viewed as morphisms and so can be suitably represented by matrices. We reduce the non-trivial invariant generation problem to the computation of the associated eigenspaces by encoding the new consecution requirements as specific morphisms represented by matrices. More specifically, we establish very general sufficient conditions that show the existence and allow the computation of invariant ideals. Our methods also embody a strategy to estimate degree bounds, leading to the discovery of rich classes of inductive, *i.e.* provable, invariants. Our approach avoids first-order quantifier elimination, Grobner basis computation or direct system resolution, thereby circumventing difficulties met by other recent techniques.

1 Introduction

Hybrid systems [1] exhibit both discrete and continuous behaviors, as one often finds when modeling digital system embedded in analog environments. Most safety-critical systems (*e.g.* aircraft, automobiles, chemicals and nuclear power plants, biological systems) operate semantically as non-linear hybrid systems. As such, they can only be adequately modeled by means of non linear arithmetic over the real numbers involving multivariate polynomials and fractional or transcendental functions. Some verification approaches for treating such models are based on the powerful Abstract Interpretation framework [2, 3] and so also on inductive invariant generation methods [4], combined with the reduction of safety-critical properties to invariant properties. More recent approaches have been constraint-based [5, 6, 7, 8, 9]. In these cases, a candidate invariant with a fixed degree and unknown parametric coefficients, *i.e.*, a template form, is proposed as the target invariant to be generated. The conditions for invariance are then encoded, resulting in constraints on the unknown coefficients

⋆ List of authors in alphabetic order.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 373–389, 2010.
© Springer-Verlag Berlin Heidelberg 2010

whose solutions yield invariants. One of the main advantage of such constraint-based approaches is that they are goal-oriented. But, on the other hand, they still require the computation of several Grobner Bases [10] or require first-order quantifier elimination [11]. But, on the other hand, known algorithms for those problems are, at least, of double exponential complexity. SAT Modulo Theory decision procedures and polynomial systems [12, 6] could also, eventually, lead to decision procedures for linear theories and decidable systems. Such ideas strive to generate linear or polynomial invariants over hybrid systems that exhibit affine or polynomial systems as continuous evolution modes. Nonetheless, despite tremendous progress over the years [5, 13, 14, 6, 8, 9, 15], the problem of invariant generation for hybrid systems remains very challenging for both non-linear discrete systems as well as non-linear differential systems with non abstracted local and initial conditions.

In this work we use hybrid automata as computational models for hybrid systems. A hybrid automaton describes the interaction between discrete transitions and continuous dynamics, the latter being governed by local differential equations. We present new methods for the automatic generation of non-linear invariants for non-linear hybrid systems. These methods give rise to more efficient algorithms, with much lower complexity in space and time. First, we extend and generalize our previous work on invariant generation for hybrid systems [16, 17]. To do so, we provide methods to generate non trivial basis of provable invariants for local continuous evolution modes described by non linear differential rules. These invariants can provide precise over-approximations of the set of reachable states in the continuous state space. As a consequence, they can determine which discrete transitions are possible and can also verify if a given property is fulfilled or not. Next, in order to generate invariants for hybrid systems we complete and extend our previous work on non linear invariant generation for discrete programs [18]. The contribution and novelty in our paper clearly differ from those in [5] as their constraint-based techniques are based on several Grobner Basis (or Syzygy Basis[19]) computations and on solving non linear problems for each location. Nevertheless, they introduce a useful formalism to treat the problem, and we start from similar definitions for hybrid systems, inductive invariants and consecution conditions.

We then propose methods to identify suitable morphisms to encode the relaxed consecution requirements. We show that the preconditions for discrete transitions and the Lie-derivatives for continuous evolutions can be viewed as morphisms over a vector space of terms, with polynomially bounded degrees, which can be suitably represented by matrices. The relaxed consecution requirements are also encoded as morphisms represented by matrices. By so doing, we do not need to start with candidate invariants that generate intractable solving problems. Moreover, our methods are not constraint-based. Rather, we automatically identify the needed degree of a generic multivariate polynomial, or fractional, as a relaxation of the consecution condition. The invariant basis are, then, generated by computing the Eigenspace of another matrix that is constructed. We identify the needed approximations and the relaxations of the

consecution conditions, in order to guaranteed sufficient conditions for the existence and computation of invariants. Moreover, the unknown parameters that are introduced are all fixed in such a way that certain specific matrices will have a non null kernel, guaranteeing a basis for non-trivial invariants.

To summarize our contributions: (i) we reduce the non-trivial invariant generation problem to the computation of associated eigenspaces; (ii) we propose a computational method of lower complexity than the mathematical foundations of previous methods (*e.g.*, Grobner basis computation, quantifier elimination, cylindrical algebraic decomposition, Syzygy basis computation); (iii) we handle non-linear hybrid systems, extended with parameters and variables that are functions of time. We note that the latter conditions are still not treated by other state-of-the-art invariant generation methods; (iv) we introduce a more general form of approximating consecution, called fractional and polynomial consecution; (v) we bring very general sufficient conditions guaranteeing the existence and allowing the computation of invariant ideals for situations not treated in the present literature on invariant generation; and (vi) our algorithm incorporates a strategy for estimating optimal degree bounds for candidate invariants, thus being able to compute basis for ideals of non-trivial non-linear invariants.

In Section 2 we introduce ideals of polynomials, inductive assertions and algebraic hybrid systems. In Section 3 we present new forms of approximating consecution for non-linear differential systems. In Section 4, we discuss morphisms suitable to handle non-linear differential rules and show how to generate invariants for differential rules. In Section 5 we introduce a strategy that can be used to choose the degree of invariants. Section 6 presents some experiments. In Section 7, we show how to generate ideals for global invariants by taking into account the ideal basis of local differential invariants, together with those derived from the discrete transition analysis and the initial constraints. We present our conclusions in Section 8.

In this writing, we strive to precede the most important proofs by sketches. Full proofs, more details and examples can be found in [20, 21].

2 Algebraic Hybrid Systems and Inductive Assertions

Let $\mathbb{K}[X_1, .., X_n]$ be the ring of multivariate polynomials over the set of variables $\{X_1, .., X_n\}$. An ideal is any set $I \subseteq \mathbb{K}[X_1, .., X_n]$ which contains the null polynomial and is closed under addition and multiplication by any element in $K[X_1, .., X_n]$. Let $E \subseteq \mathbb{K}[X_1, .., X_n]$ be a set of polynomials. The ideal generated by E is the set of finite sums $(E) = \{\sum_{i=1}^{k} P_i Q_i \mid P_i \in \mathbb{K}[X_1, \ldots, X_n], Q_i \in E, k \geq 1\}$. A set of polynomials E is said to be a *basis* of an ideal I if $I = (E)$. By the Hilbert basis theorem, we know that all ideals have a *finite basis*.

Notationally, as is standard in static program analysis, a primed symbol x' refers to next state value of x after a transition is taken. We may also write $\dot{x}$ for the derivative $\frac{dx}{dt}$. We denote by $\mathbb{R}_d[X_1, .., X_n]$ the ring of multivariate polynomials over the set of real variables $\{X_1, .., X_n\}$ of degree at most d. We write $Vect(v_1, ..., v_n)$ for the vectorial space generated by the basis $v_1, ..., v_n$.

Definition 1. *A* hybrid system *is described by a tuple* $\langle V, V_t, L, \mathcal{T}, \mathcal{C}, \mathcal{S}, l_0, \Theta \rangle$, *where* $V = \{a_1, .., a_m\}$ *is a set of parameters,* $V_t = \{X_1(t), .., X_n(t)\}$ *where* $X_i(t)$ *is a function of* t*,* L *is a set of locations and* l_0 *is the initial location. A* transition $\tau \in \mathcal{T}$ *is given by* $\langle l_{pre}, l_{post}, \rho_\tau \rangle$*, where* l_{pre} *and* l_{post} *name the pre- and post- locations of* τ*, and the* transition relation ρ_τ *is a first-order assertion over* $V \cup V_t \cup V' \cup V_t'$*. Also,* Θ *is the initial condition, given as a first-order assertion over* $V \cup V_t$*, and* $\mathcal{C}$ *maps each location* $l \in L$ *to a* local condition $\mathcal{C}(l)$ *denoting an assertion over* $V \cup V_t$*. Finally,* $\mathcal{S}$ *associates each location* $l \in L$ *to a* differential rule $\mathcal{S}(l)$ *corresponding to an assertion over* $V \cup \{dX_i/dt | X_i \in V_t\}$*. A* state *is any pair from* $L \times \mathbb{R}^{|V|}$*.* □

Definition 2. *A run of a hybrid automaton is an infinite sequence* $(l_0, \kappa_0) \rightarrow \cdots \rightarrow (l_i, \kappa_i) \rightarrow \cdots$ *of states where* l_0 *is the* initial location *and* $\kappa_0 \models \Theta$*. For any two consecutive states* $(l_i, \kappa_i) \rightarrow (l_{i+1}, \kappa_{i+1})$ *in such a run, the condition describes a* discrete consecution *if there exists a transition* $\langle q, p, \rho_i \rangle \in \mathcal{T}$ *such that* $q = l_i$*,* $p = l_{i+1}$ *and* $\langle \kappa_i, \kappa_{i+1} \rangle \models \rho_i$ *where the primed symbols refer to* κ_{i+1}*. Otherwise, it is a* continuous consecution *condition and we must have* $q \in L$*,* $\varepsilon \in \mathbb{R}$ *and a differentiable function* $\phi : [0, \varepsilon) \rightarrow \mathbb{R}^{|V \cup V_t|}$ *such that the following conditions hold: (i)* $l_i = l_{i+1} = q$*; (ii)* $\phi(0) = \kappa_i$*,* $\phi(\varepsilon) = \kappa_{i+1}$*; (iii) During the time interval* $[0, \varepsilon)$*,* ϕ *satisfies the local condition* $\mathcal{C}(q)$ *and the local differential rule* $\mathcal{S}(q)$*. That is, for all* $t \in [0, \varepsilon)$ *we must have* $\phi(t) \models \mathcal{C}(q)$ *and* $\langle \phi(t), d\phi(t)/dt \rangle \models \mathcal{S}(q)$*. A state* (ℓ, κ) *is reachable if there is a run and some* $i \geq 0$ *such that* $(\ell, \kappa) = (\ell_i, \kappa_i)$*.* □

Definition 3. *Let* W *be a hybrid system. An assertion* φ *over* $V \cup V_t$ *is an* invariant *at* $l \in L$ *if* $\kappa \models \varphi$ *whenever* (l, κ) *is a reachable state of* W*.* □

Definition 4. *Let* W *be a hybrid system and let* $\mathbb{D}$ *be an assertion domain. An assertion map for* W *is a map* $\gamma : L \rightarrow \mathbb{D}$*. We say that* γ *is* inductive *if and only if the following conditions hold:*

1. ***Initiation****:* $\Theta \models \gamma(l_0)$*;*
2. ***Discrete Consecution****: for all* $\langle l_i, l_j, \rho_\tau \rangle \in \mathcal{T}$ *we have* $\gamma(l_i) \wedge \rho_\tau \models \gamma(l_j)'$*;*
3. ***Continuous Consecution****: for all* $l \in L$*, and two consecutive states* (l, κ_i) *and* (l, κ_{i+1}) *in a possible run of* W *such that* κ_{i+1} *is obtained from* κ_i *according to the local differential rule* $\mathcal{S}(l)$*, if* $\kappa_i \models \gamma(l)$ *then* $\kappa_{i+1} \models \gamma(l)$*. Note that if* $\gamma(l) \equiv (P_\gamma(X_1(t), .., X_n(t)) = 0) \forall t \in [0, \varepsilon)$ *where* P_γ *is a multivariate polynomial in* $\mathbb{R}[X_1, .., X_n]$ *such that it has null values on the trajectory* $(X_1(t), ..., X_n(t))$ *during the time interval* $[0, \varepsilon)$ *(which do not implies that* P_γ *is the null polynomial) then* $\mathcal{C}(l) \wedge (P_\gamma(X_1(t), .., X_n(t)) = 0) \models (d(P_\gamma(X_1(t), .., X_n(t))/dt = 0)$ *during the local time interval.* □

Hence, if γ is an inductive assertion map then $\gamma(l)$ is an invariant at l for W.

3 New Continuous Consecution Conditions

Now we show how to encode differential continuous consecution conditions. Consider a hybrid automaton W. Let $l \in L$ be a location which could, eventually, be

in a circuit, and let η be an assertion map such that $\eta(l) \equiv (P_\eta(X_1(t),..,X_n(t)) = 0)$, where P_η is a multivariate polynomial in $\mathbb{R}[X_1,..,X_n]$. We have $\frac{dP_\eta}{dt} = \frac{\partial P_\eta(X_1,...,X_n)}{\partial X_1}\frac{dX_1(t)}{dt} + \cdots + \frac{\partial P_\eta(X_1,...,X_n)}{\partial X_n}\frac{dX_n(t)}{dt}$.

Definition 5. *For a polynomial P in $\mathbb{R}_d[X_1,..,X_n]$, we define the polynomial $\mathscr{D}_P$ of $\mathbb{R}_d[Y_1,..,Y_n,X_1,..,X_n]$:*

$$\mathscr{D}_P(Y_1,..,Y_n,X_1,..,X_n) = \frac{\partial P(X_1,..,X_n)}{\partial X_1}Y_1 + ... + \frac{\partial P(X_1,..,X_n)}{\partial X_n}Y_n.$$

Hence, $dP_\eta/dt = \mathscr{D}_{P_\eta}(\dot{X}_1,..,\dot{X}_n,X_1,..,X_n)$. Now, let (l,κ_i) and (l,κ_{i+1}) be two consecutive configurations in a run. Then we can express local state continuous consecutions as $\mathcal{C}(l) \wedge (P_\eta(X_1(t),..,X_n(t)) = 0) \models (dP_\eta(X_1(t),..,X_n(t)/dt = 0)$.

Definition 6. *Let W be a hybrid automaton, $l \in L$ a location and let η be an algebraic inductive map with $\eta(l) \equiv (P_\eta(X_1(t),..,X_n(t)) = 0)$ for all t in the time interval of mode l (so, P_η has a null value over the local trajectory $(X_1(t),..,X_n(t))$). We identify the following notions to encode* continuous consecution conditions*:*

– *η satisfies a differential* Polynomial-*scale consecution at l if and only if there exist a multivariate polynomial T such that $\mathcal{C}(l) \models dP_\eta/dt - TP_\eta = 0$. We say that P_η is a polynomial-scale and a T-scale differential invariant.*

Differential Polynomial-scale consecution encode the fact that the numerical value of the Lie derivative of the polynomial P_η associated with assertion $\eta(l)$ is given by T times its numerical value throughout the time interval $[0,\varepsilon]$. In [16, 17] we proposed methods for T-scale invariant generation where T is a constant (constant-scaling) or null (strong-scaling). As can be seen, the consecution conditions are relaxed when going from strong to polynomial scaling. Also, the T polynomials or can be understood as *template multiplicative factors*. In other words, they are polynomials with unknown coefficients. In the next section, we consider polynomial-scale consecution and then we could extend the methods to fractional-scale consecution conditions, as is done in Section 7 for discrete steps. In later sections we show how to combine these conditions with others induced by discrete transitions. In [20], [21] one can find more details on how to handle other constraints associated to locations.

Theorem 1. *(Soundness) Let P be a continuous function and let $S = [\dot{X}_1(t) = P_1(X_1(t),..,X_n(t)),..,\dot{X}_n(t) = P_n(X_1(t),..,X_n(t)]$ be a differential rule, with initial condition $(x_1,..,x_n)$. Any polynomial which is a P-scale differential invariant for these initial conditions is actually an inductive invariant.*

Consider polynomial-scale consecution, the system $[\dot{x} = ax(t);\ \dot{y} = ay(y) + bx(t)y(t)]$ could be cited as counter-example for completeness as its invariants are not P-scale differential invariant.

4 Handling Non-linear Differential Systems

Invariant generation for continuous time evolution is one of the main challenging step in static analysis and verification of hybrid systems. That is why we first restrict the analysis to differential system which appear in locations. We consider a

non linear differential system of the form: $[\dot{X}_1(t) = P_1(X_1(t), .., X_n(t)); ..; \dot{X}_n(t) = P_n(X_1(t), .., X_n(t))]$, with the P_i's in $\mathbb{R}[X_1, .., X_n]$.We have the following lemma.

Lemma 1. *Let $Q \in \mathbb{R}[X_1, .., X_n]$ such that $\mathscr{D}_Q(P_1, .., P_n, X_1, .., X_n) = TQ$ with T in $\mathbb{R}[X_1, .., X_n]$. Then Q is a T-scale invariant.* □

If $P \in \mathbb{R}[X_1, .., X_n]$ is of degree r and the maximal degree of the P_i's is d, then the degree of $\mathscr{D}_P(P_1, .., P_n, X_1, .., X_n)$ is $r + d - 1$. Hence, T must be searched in the subspace of $\mathbb{R}[X_1, .., X_n]$, which is of degree at most $r + d - 1 - r = d - 1$. Transposing the situation to linear algebra, consider the morphism

$$D : \begin{cases} \mathbb{R}_r[X_1, \ldots, X_n] \to \mathbb{R}_{r+d-1}[X_1, \ldots, X_n] \\ \qquad P \mapsto \mathscr{D}_P(P_1, \ldots, P_n, X_1, \ldots, X_n). \end{cases}$$

Let M_D be its matrix in the canonical basis of $\mathbb{R}_r[X_1, ., X_n]$ and $\mathbb{R}_{r+d-1}[X_1, .., X_n]$.

Example 1. Consider the following differential rules: $[\dot{x} = x^2 + xy + 3y^2 + 3x + 4y + 4 \ ; \ \dot{y} = 3x^2 + xy + y^2 + 4x + y + 3\ ]$. In this example we write $P_1(x, y) = x^2 + xy + 3y^2 + 3x + 4y + 4$ and $P_2(x, y) = 3x^2 + xy + y^2 + 4x + y + 3$. We consider the associated morphism D from $\mathbb{R}_2[x, y]$ to $\mathbb{R}_3[x, y]$. Using the basis $B_1 = (x^2, xy, y^2, x, y, 1)$ of $\mathbb{R}_2[x, y]$ and $B_2 = (x^3, x^2y, xy^2, y^3, x^2, xy, y^2, x, y, 1)$ of $\mathbb{R}_3[x, y]$, we define the matrix M_D. To do so, we compute $D(P)$ for all elements P in the basis $(x^2, xy, y^2, x, y, 1)$ and we express the results in the basis $(x^3, x^2y, xy^2, y^3, x^2, xy, y^2, x, y, 1)$. Hence, to get the first column of M_D we first consider $P(x, y) = x^2$, the first element of B_1, and we compute $D(P) = \mathscr{D}_P(P_1, P_2, x, y)$ which is expressed in B_2 as

$$D(x^2) = 2\,x^3 + 2\,x^2y + 6\,xy^2 + 0\,y^3 + 6\,x^2 + 8\,xy + 0\,y^2 + 8\,x + 0\,y + 0 \times 1$$

$$M_D = \begin{pmatrix} 2 & 2 & 6 & 0 & 6 & 8 & 0 & 8 & 0 & 0 \\ 3 & 2 & 2 & 3 & 4 & 7 & 4 & 3 & 4 & 0 \\ 0 & 6 & 2 & 2 & 0 & 8 & 2 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 3 & 3 & 4 & 4 \\ 0 & 0 & 0 & 0 & 3 & 1 & 1 & 4 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^{\top}.$$

□

Choosing a generic T in $\mathbb{R}_{d-1}[X_1, .., X_n]$, we define the associated morphism

$$\overline{T} : \begin{cases} \mathbb{R}_r[x_1, \ldots, x_n] \to \mathbb{R}_{r+d-1}[x_1, \ldots, x_n] \\ \qquad P \mapsto TP. \end{cases}$$

Denote by L_T its matrix in the canonical basis, obtained as in the computation of M_D. If T is a *generic template* in $\mathbb{R}_{d-1}[X_1, .., X_n]$, call $t_1, .., t_{v(d-1)}$ its coefficients where $v(d-1)$ is the dimension of $\mathbb{R}_{d-1}[X_1, .., X_n]$. Then L_T's coefficients are in $\{t_1, .., t_{v(d-1)}\}$ and it has a natural block decomposition. We will call $M(pol)$ the set of such matrices. Now let $Q \in \mathbb{R}[X_1, .., X_n]$ be a T-scale invariant for a given differential system defined by $P_1, .., P_n \in \mathbb{R}[X_1, .., X_n]$. Then $(\mathscr{D}_Q(P_1, .., P_n, X_1, .., X_n) = TQ) \Leftrightarrow D(Q) = \overline{T}(Q) \Leftrightarrow ((D - \overline{T})(Q) = 0_{\mathbb{R}[X_1, .., X_n]}) \Leftrightarrow (Q \in Ker(M_D - L_T))$. So, a T-scale invariant is nothing else than a vector in the kernel of $M_D - L_T$.

Theorem 2. *There is a polynomial-scale invariant for the differential system if and only if there exists a matrix L_T in $M(pol)$, corresponding to a polynomial T of $\mathbb{R}_{d-1}[x_1,..,X_n]$, such that $Ker(M_D - L_T)$ is not reduced to zero. And, any vector in the kernel of $M_D - L_T$ will give a T-scale differential invariant.* □

Now notice that $M_D - L_T$ with a non trivial kernel is equivalent to it having rank strictly less than the dimension $v(r)$ of $\mathbb{R}_r[x_1,\ldots,x_n]$. By a classical theorem [22], this is equivalent to the fact that each $v(r) \times v(r)$ sub-determinant of $M_D - L_T$ is equal to zero. Those determinants are polynomials with variables $(t_1,..,t_{v(d-1)})$, which we will denote by $E_1(t_1,...,t_{v(d-1)}),...,E_s(t_1,...,t_{v(d-1)})$.

Theorem 3. *There is a non trivial T-scale invariant if and only if the polynomials $(E_1,..,E_s)$ admit a common root, other than the trivial one $(0,...,0)$.* □

This theorem provides us with important existence results. But there is a more practical way to get invariant ideals without computing common roots. Consider initial values given by unknown parameters $(x_1(0) = u_1,\ldots,x_n(0) = u_n)$. The initial step defines a linear form on $\mathbb{R}_r[x_1,\ldots,x_n]$, namely $I_u : P \mapsto P(u_1,...,u_n)$. Hence, initial values correspond to a hyperplane of $\mathbb{R}_r[X_1,..,X_n]$ given by the kernel I_u, which is $\{Q \in \mathbb{R}_r[X_1,..,X_n] | Q(u_1,\ldots,u_n) = 0\}$.

Theorem 4. *Let Q be in $\mathbb{R}_r[X_1,..,X_n]$. Then Q is an inductive invariant for the differential system with initial values $(u_1,..,u_n)$ if and only if there exists a matrix $L_T \neq 0$ in $M(pol)$, corresponding to T in $\mathbb{R}_{d-1}[X_1,..,X_n]$, such that Q is in the intersection of $Ker(M_D - L_T)$ and the hyperplane $Q(u_1,\ldots,u_n) = 0$.* □

Now, if $Dim(Ker(M_D - L_T)) \geq 2$ then $Ker(M_D - L_T)$ would intersect any initial (semi-)hyperplane.

Corollary 1. *There are non-trivial invariants for any given initial values if and only if there exists a matrix L_T in $M(pol)$ such that $Ker(M_D - L_T)$ has dimension at least 2.* □

Also, we have $(Dim(Ker(M_D - L_T)) \geq 2)$ if and only if we also have $Rank(M_D - L_T) \leq Dim(\mathbb{R}_r[X_1,..,X_n]) - 2$. Further, we also show how to assign values to the coefficients of T in order to guarantee the existence and generation of invariants.

Example 2. (Running example) Consider the following differential rules with $P_1 = x^2 + 2xy + x$ and $P_2 = xy + 2y^2 + y$:

$$[\ \dot{x}(t) = x^2(t) + 2x(t)y(t) + x(t)\ ;\ \dot{y}(t) = x(t)y(t) + 2y^2(t) + y(t)\ ]. \qquad (1)$$

Step 1: We build matrix $M_D - L_T$. The maximal degree of the systems is $d = 2$ and the T-scale invariant will be of degree $r = 2$. Then, T is of degree $d - 1 = 1$ and we write t_1, t_2, t_3 for its unknown coefficients, *(i.e.* the canonical form is $T(x,y) = t_1x + t_2y + t_3$. Using the basis $(x^2, xy, y^2, x, y, 1)$ of $\mathbb{R}_2[x,y]$ and the basis $(x^3, x^2y, xy^2, y^3, x^2, xy, y^2, x, y, 1)$ of $\mathbb{R}_3[x,y]$, the matrix $M_D - L_T$ is:

$$M_D - L_T = \begin{pmatrix} 2-t_0 & 0 & 0 & 0 & 0 & 0 \\ 4-t_1 & 2-t_0 & 0 & 0 & 0 & 0 \\ -t_2 & 4-t_1 & 2-t_0 & 0 & 0 & 0 \\ 0 & 0 & 4-t_1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1-t_0 & 0 & 0 \\ 0 & 2-t_2 & 0 & 2-t_1 & 1-t_0 & 0 \\ 0 & 0 & 2-t_2 & 0 & 2-t_1 & 0 \\ 0 & 0 & 0 & 1-t_2 & 0 & -t_0 \\ 0 & 0 & 0 & 0 & 1-t_2 & -t_1 \\ 0 & 0 & 0 & 0 & 0 & -t_2 \end{pmatrix}.$$

Step 2: Now the unknown t_i's are given values so as to guarantee the existence of invariants. Our algorithm proposes to fix $t_1 = 2$, $t_2 = 4$ and $t_3 = 2$ to get $T(x,y) = 2x + 4y + 2$. Matrix $M_D - L_T$ has its second and third columns equal to zero. So, the rank of $M_D - L_T$ is less than 4 and its kernel has dimension at least 2. Any vector in this kernel will be a T-scale differential invariant.

Step 3: Now, Corollary 1 applies to $M_D - L_T$. So, there will always be invariants, whatever the initial values. We compute and output the basis of $Ker(M_D - L_T)$:

```
Polynomial scaling continuous evolution
T(x,y) = 2 x + 4 y + 2
Module of degree 6 and rank 2 and Kernel of dimension 4
{{0, 1, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0}}
```

The vectors of the basis are interpreted in the canonical basis of $\mathbb{R}_2[x,y]$:

```
Basis of invariant Ideal
{x y, y^2}
```

We have an ideal for non trivial inductive invariants and we search for one of the form $axy + by^2$. If the system has initial conditions $x(0) = \lambda$ and $y(0) = \mu$, then $a\lambda\mu + b\mu^2 = 0$, and $\mu xy - \lambda y^2 = 0$ is an invariant for all μ and λ. □

5 Obtaining Optimal Degree Bounds

In order to guarantee the existence of non-trivial invariants of degree r, we need a polynomial T such that $Ker(M_D - L_T) \neq 0$. First, define T as a polynomial with parameterized coefficients. We can then build a decision procedure to assign values to the coefficients of T in such a way that $Ker(M_D - L_T) \neq 0$. The pseudo code depicted in Algorithm 1 illustrates this strategy. Algorithm 1 is in a standard form, but its contribution relies on very general sufficient conditions for the existence and the computation of invariants. From the differential rules, we obtain matrix M_D (see line 8) with real entries. We can then define degree bounds for matrices L_T that can be used to approximate the consecution requirements (see line 9). As we recall from Section 4, $Ker(M_D - L_T) \neq 0$ is equivalent to having $M_D - L_T$ with rank strictly less than the dimension $v(r)$ of $\mathbb{R}_r[x_1, \ldots, x_n]$. We then reduce the rank of $M_D - L_T$ by assigning values of terms in M_D to parameters in L_T (see line 10). Next, we determine whether the obtained matrix $\bar{M}$ has a trivial kernel by first computing its rank and then checking

Algorithm 1. Ideal_Inv_Gen$(r, P_1, ..., P_n, X_1, ..., X_n)$

```
    /*Guessing the degree bounds.*/
    Data: r is the degree for the set of invariants we are looking for, P1,..Pn are
          the n polynomials given by the considered differential rules, and
          X1,..Xn ∈ Vt are functions of time.
    Result: B_Inv, a basis of ideal of invariants.
    begin
 1      int d
 2      Template T
 3      Matrix M_D, L_T
 4      d ⟵ Max_degree({P1, ..., Pn})
 5      /*d is the maximal degree of Pi's*/
 6      if d >= 2 then
 7          T ⟵ Template_Canonical_Form(d − 1)
 8          M_D ⟵ Matrix_D(r, r + d − 1, P1, ..., Pn)
 9          L_T ⟵ Matrix_L(r, r + d − 1, T)
10          M̄ ⟵ Reduce_Rank_Assigning_Values(M_D − L_T)
11          if Rank(M̄) >= Dim(R_r[X1, .., Xn]) then
12              return Ideal_Inv_Gen(r + 1, P1, ..., Pn, X1, ..., Xn)
13              /*We need to increase the degree r of candidates invariants.*/
14          else
15              return Nullspace_Basis(M̄)
16              /*There exists an ideal of invariants that we can compute*/
17      else
18          ... /*We refer to our previous work for strong and constant scaling.*/
    end
```

if $(\mathbf{Rank}(\bar{M}) < \mathbf{Dim}(R_r[X_1, .., X_n]))$ holds (see line 11). By so doing, we can increase the degree r of invariants until Theorem 2 (or Corollary 1) applies or until stronger hypotheses occur, *e.g.* if all $v(r) \times v(r)$ sub-determinants are null. Then, we compute and output the basis of the nullspace of matrix $\bar{M}$ in order to construct an ideal basis for non trivial invariants (see `Nullspace_Basis`, line 15). We can directly see that if there is no ideal for non-trivial invariants for a value r_i then we conclude that there is no ideal of non-trivial invariants for all degrees $k \leq r_i$. This could guide other constraint-based techniques, since checking for invariance with a template of degree less or equal to r_i will not be necessary. In case there is no ideal for invariants of degree r (see line 12), we first increment the value of r by 1 before the recursive call to `Ideal_Inv_Gen`.

We thus showed how to reduce the invariant generation problem to the problem of computing a kernel basis for polynomial mappings. For the latter, we use well-known state-of-the-art algorithms, *e.g.* that Mathematica provides. These algorithms calculate the eigenvalues and associated eigenspaces of $\bar{M}$ when it is a square matrix. When $\bar{M}$ is a rectangular matrix, we can use its *singular value decomposition* (SVD). A SVD of $\bar{M}$ provides an explicit representation of its

rank and kernel by computing unitary matrices U and V and a regular diagonal matrix S such that $\bar{M} = USV$. We compute the SVD of a $v(r+d-1) \times v(r)$ matrix $\bar{M}$ by a two step procedure. First, reduce it to a bi-diagonal matrix, with a cost of $O(v(r)^2 v(r+d-1))$ flops. The second step relies on an iterative method, as is also the case for other eigenvalue algorithms. In practice, however, it suffices to compute the SVD up to a certain precision, *i.e.* up to a machine epsilon. In this case, the second step takes $O(v(r))$ iterations, each using $O(v(r))$ flops. So, the overall cost is $O(v(r)^2 v(r+d-1))$ flops. For the implementation of the algorithm we could rewrite Corollary 1 as follow.

Corollary 2. *Let $\bar{M} = U \cdot S \cdot V$ be the singular value decomposition of matrix $\bar{M}$ described just above. There will be a non trivial T-invariant for any given initial condition if and only if the number of non-zero elements in matrix S is less than $v(r) - 2$, where $v(r)$ is the dimension of $\mathbb{R}_r[x_1, \ldots, x_n]$. Moreover, the orthonormal basis for the nullspace obtained from the decomposition directly gives an ideal for non-linear invariants.*

It is important to emphasize that eigenvectors of $\bar{M}$ are computed after the parameters of L_T have been assigned. When the differential system has several variables and none or few parameters, $\bar{M}$ will be over the reals and there will be no need to use the symbolic version of these algorithms.

6 Examples and Experimental Results

By reducing the problem to Linear Algebra, we are able to combine it with new optimization techniques, as illustrated in the following examples. Depending on the form of the monomials present in the system, we may be able to find T and a vector X such that $X \in Ker(M_D - L_T)$ without defining T as a template, *i.e.* without using a polynomial with unknown coefficients for scaling consecution. The idea is to directly obtain a suitable T by *factorization*. For instance, we can identify the following large classes of systems where the methods apply.

Example 3. Let $s \in \mathbb{N}$ be a positive and consider the following differential rules:

$$\begin{bmatrix} \dot{x}_1(t) = \sum_{k=0}^{s} a_k x_1(t)^{k+1} x_2(t)^k \cdots x_n(t)^k \\ \vdots \\ \dot{x}_n(t) = \sum_{k=0}^{s} a_k x_1(t)^k \cdots x_{n-1}(t)^k x_n(t)^{k+1} \end{bmatrix}. \tag{2}$$

This differential system contains parameters and variables that are time functions. We denote the polynomials thus $P_1 = \sum_{k=0}^{s} a_k x_1^{k+1} x_2^k \ldots x_n^k; \quad \ldots \quad ; P_n = \sum_{k=0}^{s} a_k x_1^k \ldots x_{n-1}^k x_n^{k+1}$. Let D be the morphism associated with (2) and let M_D be its matrix in the canonical basis. Then, it is immediate that $\mathscr{D}_P(x_i) = P_i$. Now, for this particular class of P_i's, we see that $\mathscr{D}_P(x_i) = x_i T$, where $T = \sum_{k=0}^{s} a_k x_1^k x_2^k \ldots x_{n-1}^k x_n^k$. This means that if $\overline{T}$ is the morphism associated to multiplication by T, we have $\mathscr{D}_P(x_i) = \overline{T}(x_i)$ for each i. Let L_T be its matrix in the canonical basis. We deduce that $Vect(x_1, .., x_n) \subset Ker(M_D - L_T)$. Hence,

Table 1. Examples and Experimental Results

(a) Linear algebraic problems and consecution approximations

Aprox.Consec.	Lin. Algeb. Prob.	Existence Conditions
Strong	nullspaces	$Ker(M_D) \neq \emptyset$ or (see [16]) $\exists(Q_1,..,Q_n) \in Syz(P_1,..,P_n)$, s.t $\partial_i Q_j = \partial_j Q_i$
Lambda	eigenspaces	$Ker(M_D) \geq 2$ for any init. cond., and $Ker(M_D) \neq \emptyset$ otherwise.
Polynomial	nullspaces	$Ker(M_D - L_T) \geq 2$ for any init. cond., and $Ker(M_D - L_T) \neq \emptyset$ otherwise.

(b) Experimental results: Basis of invariant ideals obtained automatically by our prototype Ideal_Inv_Gen written in Mathematica

Differential Syst.	Scal.	CPU/s
From [20] Example 6.	*Poly.*	1.12
From [20] Example 6, system (9).	*Pol.*	0.04
From [17] Example 1, system (3).	*Poly.*	0.34
Example 3, systems 2.	*Poly.*	98.49
From [20] Example 7, system (11).	*Poly.*	0.43
From [20] Example 8.	*Lamb.*	2.48
From [20] Example 2, system (6).	Str.	0.02
Human Blood Glucose Metabolism (Type I diabetic patient) [6]	Lamb.	0.04
[15, 23], Example 4, air traffic management systems.	Str.	1.29
[20] Example 4, system (8).	Lamb.	0.03
[20, 16], Generalization to dimension n of the rotational motion of a rigid body.	Str.	15.90
Example 4, system 4.	Str.	1.04

for $n \geq 2$, the space $Ker(M_D - L_T)$ has dimension greater than 2, and we can apply our existence theorem for invariants, given any initial values. We can then search for an invariant of the form $a_1x_1 + \cdots + a_nx_n$. Given the initial conditions $(x_1(0) = \lambda_1, \ldots, x_n(0) = \lambda_n)$, a vector $(a_1 \cdots a_n)^\top$ is such that the polynomial $a_1x_1 + \cdots + a_nx_n$ is an invariant for (2) whenever it belongs to the kernel of the linear form with matrix $(\lambda_1, \ldots, \lambda_n)$. Summarizing, with polynomial scaling, any polynomial $Q = a_1x_1 + \cdots + a_nx_n$ with $(a_1 \cdots a_n)^\top$ in the kernel of $(\lambda_1, \ldots, \lambda_n)$ is an invariant for (2). □

Example 4. In order to handle air traffic management systems [15, 23] automatically, we consider the given differential system:

$$[\ \dot{x_1} = a_1 cos(\omega t + c)\ ;\ \dot{x_2} = a_2 sin(\omega t + c)\ ]. \tag{3}$$

This models the system satisfied by one of the two airplanes. We introduce the new variables d_1 and d_2 to handle the transcendental functions, axiomatizing them by differential equations, so that d_1 and d_2 satisfy

$$[\ \dot{d_1} = -a_1/a_2\omega d_2\ ;\ \dot{d_2} = a_2/a_1\omega d_1\ ]. \tag{4}$$

If D is the morphism associated to this system, it is immediate that $D(a_2^2d_1^2) = -2a_1a_2\omega d_1d_2$ whereas $D(a_1^2d_2^2) = 2a_1a_2\omega d_1d_2$. From [16, 17], it implies that $Vect(a_2^2d_1^2 + a_1^2d_2^2) \subset Ker(D)$ and so $a_2^2d_1^2 + a_1^2d_2^2$ is a strong-scale invariant (*i.e.* a T-scale invariant where T is null) for the system. But $\dot{x_1} = d_1 = [a_1/(a_2\omega)]\dot{d_2}$ and $\dot{x_2} = d_2 = [-a_2/(a_1\omega)]\dot{d_1}$. Therefore, there exist constants c_1 and c_2, determined

by the initial values, such that $x_1 = a_1/a_2\omega d_2 + c_1$ and $x_2 = d_2 = -a_2/a_1\omega d_1 + c_2$. This implies that $(a_2x_1 - k_1)^2 + (a_1x_2 - k_2)^2 = 0$, with $k_1 = a_2c_1$ and $k_2 = a_1c_2$, is an invariant of the first system. Hence the two airplanes, at least for some lapse of time, follow an elliptical path. □

Table 1(a) summarizes the type of linear algebraic problems associated with each consecution approximation. The last column gives some existential results that could be reused by any constraint-based approach or reachability analysis. In Table 1(b) we list some experimental results.

7 Handling Algebraic Discrete Transition Systems

In this section we treat discrete transitions by extending and adapting our previous work on loop invariant generation for discrete programs [21, 18]. We also consider discrete transitions that are part of connected components and circuits, thus generalizing the case for simple propagation. We recall that V_k denotes the subspace of $\mathbb{R}[X_1, .., X_n]$ of degree at most k.

Definition 7. *Let $\tau = \langle l_i, l_j, \rho_\tau \rangle$ be a transition in $\mathcal{T}$ and let η be an algebraic inductive map with $\eta(l_i) \equiv (P_\eta(X_1, .., X_n) = 0)$ and $\eta(l_j) \equiv (P'_\eta(X_1, .., X_n) = 0)$.*

- *We say that η satisfies a* Fractional-scale consecution for τ *if and only if there exists a multivariate fractional $\frac{T}{Q}$ such that $\rho_\tau \models (P_\eta(X'_1, .., X'_n) - \frac{T}{Q} P_\eta(X_1, .., X_n) = 0)$. We also say that P_η is a $\frac{T}{Q}$-scale discrete invariant.*
- *We say that η satisfies a* Polynomial-scale consecution for τ *if and only if there exists a multivariate polynomial T such that $\rho_\tau \models (P_\eta(X'_1, .., X'_n) - TP_\eta(X_1, .., X_n) = 0)$. We also say that P_η is a polynomial-scale and a T-scale discrete invariant.*

7.1 Discrete Transition with Polynomial Systems

Consider an algebraic transition system: $\rho_\tau \equiv [X'_1 = P_1(X_1, .., X_n), ..., X'_n = P_n(X_1, .., X_n)]$, where the P_i's are in $\mathbb{R}[X_1, .., X_n]$. We have the following T-scale discrete invariant characterization.

Theorem 5. *A polynomial Q in $\mathbb{R}[X_1, .., X_n]$ is a T-scale discrete invariant for polynomial-scale consecution with* parametric polynomial $T \in \mathbb{R}[X_1, ..., X_n]$ *for τ if and only if $Q(P_1(X_1, .., X_n), .., P_n(X_1, .., X_n)) = T(X_1, .., X_n)Q(X_1, .., X_n)$.*

If $Q \in \mathbb{R}[X_1, .., X_n]$ is of degree r and the maximal degree of the P_i's is d, then we are looking for a T of degree $e = dr - r$. Write its ordered coefficients as $\lambda_0, ..., \lambda_s$, with $s+1$ being the number of monomials of degree inferior to e. Let M be the matrix, in the canonical basis of V_r and V_{dr}, of the morphism $\mathscr{M}$ from V_r to V_{dr} given by $Q(X_1, .., X_n) \mapsto Q(P_1(X_1, .., X_n), .., P_n(X_1, .., X_n))$. Let L be the matrix, in the canonical basis of V_r and V_{dr}, of the morphism $\mathscr{L}$ from V_r to V_{dr} given by $P \mapsto TP$. Matrix L will have a very simple form: its non zero coefficients are the λ_i's, and it has a natural block decomposition. Now let $Q \in$

$\mathbb{R}[X_1,..,X_n]$ be a T-scale discrete invariant for a transition relation defined by the P_i's. Then $Q(P_1(X_1,..,X_n),..,P_n(X_1,..,X_n)) = T(X_1,..,X_n)Q(X_1,..,X_n) \Leftrightarrow \mathscr{M}(Q) = \mathscr{L}(Q) \Leftrightarrow (\mathscr{M} - \mathscr{L})(Q) = 0_{\mathbb{R}[X_1,..,X_n]}) \Leftrightarrow (Q \in Ker(M - L))$. A T-scale discrete invariant is nothing else than a vector in the kernel of $M - L$. Our problem is equivalent to finding a L such that $M - L$ has a non trivial kernel.

Theorem 6. *Consider M as described above. Then, (i) there will be a T-scale discrete invariant if and only if there exists a matrix L (corresponding to $P \mapsto TP$) such that $M - L$ has a nontrivial kernel. Further, any vector in the kernel of $M - L$ will give a T-scale invariant polynomial; (ii) there will be a non trivial inductive invariant if and only if there exists a matrix L such that the intersection of the kernel of $M - L$ and the hyperplane given by the initial values is not zero. The invariants correspond to vectors in the intersection; and (iii) if $dim(Ker(M - L)) \geq 2$, then the basis of $Ker(M - L)$ is a basis for non trivial inductive invariants, whatever the initial conditions.*

Example 5. (Running example) Let's consider the following transition:

$$\tau = \langle l_i, l_j, \rho_\tau \equiv [\ x' = xy + x\ ;\ y' = y^2\ ]\rangle.$$

Step 1: We build matrix $M - L$. The maximal degree of the system ρ_τ is $d = 2$ and the T-scale invariant will be of degree $r = 2$. Then, T is of degree $e = dr - r = 2$ and we write $\lambda_0, ..., \lambda_5$ as its ordered coefficients *i.e.* its canonical form is $T = \lambda_0 x^2 + \lambda_1 xy + \lambda_2 y^2 + \lambda_3 x + \lambda_4 y + \lambda_5$. Consider the associated morphisms $\mathscr{M}$ and $\mathscr{L}$ from $\mathbb{R}_2[x, y]$ to $\mathbb{R}_4[x, y]$. Using the basis $C_1 = (x^2, xy, y^2, x, y, 1)$ of $\mathbb{R}_2[x, y]$ and the basis $C_2 = (x^4, yx^3, y^2x^2, y^3x, y^4, x^3, x^2y, xy^2, y^3, x^2, xy, y^2, x, y, 1)$ of $\mathbb{R}_4[x, y]$, our algorithm compute the matrix $M - L$ as

$$M - L = \begin{pmatrix} -\lambda_0 & 0 & 0 & 0 & 0 & 0 \\ -\lambda_1 & -\lambda_0 & 0 & 0 & 0 & 0 \\ 1-\lambda_2 & -\lambda_1 & -\lambda_0 & 0 & 0 & 0 \\ 0 & 1-\lambda_2 & -\lambda_1 & 0 & 0 & 0 \\ 0 & 0 & 1-\lambda_2 & 0 & 0 & 0 \\ -\lambda_3 & 0 & 0 & -\lambda_0 & 0 & 0 \\ 2-\lambda_4 & -\lambda_3 & 0 & -\lambda_1 & -\lambda_0 & 0 \\ 0 & 1-\lambda_4 & -\lambda_3 & -\lambda_2 & -\lambda_1 & 0 \\ 0 & 0 & -\lambda_4 & 0 & -\lambda_2 & 0 \\ 1-\lambda_5 & 0 & 0 & -\lambda_3 & 0 & -\lambda_0 \\ 0 & -\lambda_5 & 0 & 1-\lambda_4 & -\lambda_3 & -\lambda_1 \\ 0 & 0 & -\lambda_5 & 0 & 1-\lambda_4 & -\lambda_2 \\ 0 & 0 & 0 & 1-\lambda_5 & 0 & -\lambda_3 \\ 0 & 0 & 0 & 0 & -\lambda_5 & -\lambda_4 \\ 0 & 0 & 0 & 0 & 0 & 1-\lambda_5 \end{pmatrix}.$$

Step 2: We then reduce the rank of $M - L$ by assigning values to the λ_i's. Our procedure fixes $\lambda_0 = \lambda_1 = \lambda_3 = 0$, $\lambda_2 = \lambda_5 = 1$ and $\lambda_4 = 2$, so that $T(x, y) = y^2 + 2y + 1$. The first column of $M - L$ becomes zero and the second column is equal to the fourth. Hence, the rank of $M - L$ is less than 4 and its kernel has dimension at least 2. Any vector in this kernel will be a T-invariant.

Step 3: Now matrix $M - L$ satisfies the hypotheses of Theorem 6(iii). So, there will always be invariants, whatever the initial values. We compute the basis of $Ker(M - L)$:

```
Polynomial scaling discrete step
T(x,y) = y^2 + 2 y + 1
Module of degree 6 and rank 3 and Kernel of dimension 3
{{1, 0, 0, 0, 0, 0}, {0, 1, 0, -1, 0, 0}, {0, 0, 1, 0, -2, 1}}
```

The vectors of the basis are interpreted in the canonical basis C_1 of $\mathbb{R}_2[x, y]$:

```
Basis of invariant Ideal
{x^2, x y - x, y^2 - 2 y + 1}
```

We thus obtained an ideal of non trivial inductive invariants. In other words, for all $G_1, G_2, G_3 \in \mathbb{R}[x, y]$, $(G_1(x, y)(x^2)+G_2(x, y)(xy-x)+G_3(x, y)(y^2-2y+1) = 0)$ is an inductive invariant. For instance, consider the initial step $(y = y_0, x = 1)$. A possible invariant is $y_0(1 - y_0)x^2 + xy - x + y^2 - 2y + 1 = 0$. □

7.2 Discrete Transition with Fractional Systems

We now want to deal with transition systems ρ_τ of the following type:

$$[X_1' = P_1(X_1, .., X_n)/Q_1(X_1, .., X_n), .., X_n' = P_n(X_1, .., X_n)/Q_n(X_1, .., X_n)],$$

where the P_i's and Q_i's belong to $\mathbb{R}[X_1, .., X_n]$ and P_i is relatively prime to Q_i. One need to relax the consecution conditions to fractional-scale as soon as fractions appear in the transition relation.

Theorem 7. *(F-scale invariant characterization) A polynomial Q in $\mathbb{R}[X_1, .., X_n]$ is a F-scale invariant for fractional discrete scale consecution with a* parametric fractional $F \in \mathbb{R}(X_1, .., X_n)$ *for τ if and only if* $Q\left(\frac{P_1}{Q_1}, .., \frac{P_n}{Q_n}\right) = FQ$.

Let d be the maximal degree of the P_i's and Q_i's, and let Π be the least common multiple (lcm) of the Q_i's. Further, suppose that we are looking for a F-invariant Q of degree r. Let $\mathscr{M}$ be the morphism of vector spaces $Q \mapsto \Pi^r Q(P_1/Q_1, .., P_n/Q_n)$ from V_r to V_{nrd}, and let M be its matrix in a canonical basis. Let T be a polynomial in V_{nrd-r}, let $\mathscr{L}$ denote the morphism of vector spaces $Q \mapsto TQ$ from V_r to V_{nrd}, with L its matrix in a canonical basis. As we show in the following theorem, our problem is equivalent to finding a L such that $M - L$ has a non trivial kernel.

Theorem 8. *Consider M and L as described above. Then, (i) there exists F-scale invariants (with F is of the form T/Π^r) if and only if there exists a matrix L such that $Ker(M - L) \neq \emptyset$. In this situation, any vector in the kernel of $M - L$ will give a F-scale discrete invariant; (ii) we have a non trivial invariant if and only if there exists a matrix L such that the intersection of the kernel of $M - L$ and the hyperplane given by the initial values is not zero, the invariants corresponding to vectors in the intersection; and (iii) we will have a non-trivial invariant for any non-trivial initial value if there exists a matrix L such that the dimension of $Ker(M - L)$ is at least* 2.

Example 6. Consider the system

$$\rho_\tau \equiv [\ x_1' = x_2/(x_1+x_2)\ ;\ x_2' = x_1/(x_1+2x_2)\ ].$$

We are looking for a F-scale invariant polynomial of degree two. The lcm of (x_1+x_2) and (x_1+2x_2) is their product, so that $\mathscr{M}$ is given by: $[Q \in V_2 \mapsto [(x_1+x_2)(x_1+2x_2)]^2 Q(x_1/(x_1+x_2), x_2/(x_1+2x_2))]$. As both $x_2/(x_1+x_2)$ and $x_1/(x_1+2x_2)$ have "degree" zero, $[(x_1+x_2)(x_1+2x_2)]^2 Q(x_2/(x_1+x_2), x_1/(x_1+2x_2))$ will be a linear combination of degree four, if it is non null. Hence, $\mathscr{M}$ has values in $Vect(x_1^4, x_1^3x_2, x_1^2x_2^2, x_1x_2^3, x_2^4)$. For T and Q in V_2 to verify $[(x_1+x_2)(x_1+2x_2)]^2 Q(x_2/(x_1+x_2), x_1/(x_1+2x_2)) = TQ$, as the left member is in $Vect(x_1^4, x_1^3x_2, x_1^2x_2^2, x_1x_2^3, x_2^4)$, T must be of the form $\lambda_0 x_1^2 + \lambda_1 x_1x_2 + \lambda_2 x_2^2$ and Q of the form $a_0x_1^2 + a_1x_1x_2 + a_3x_2^2$. We see that we can take Q in $Vect(x_1^2, x_1x_2, x_2^2)$, and similarly for T. Then both $\mathscr{M}$, $\mathscr{L}$: $(Q \mapsto TQ)$ will be morphisms from $Vect(x_1^2, x_1x_2, x_2^2)$ in $Vect(x_1^4, x_1^3x_2, x_1^2x_2^2, x_1x_2^3, x_2^4)$. In the corresponding canonical basis, the matrix $M - L$ is

$$M - L = \begin{pmatrix} -\lambda_0 & 0 & 1 \\ -\lambda_1 & 1-\lambda_0 & 2 \\ 1-\lambda_2 & 3-\lambda_1 & 1-\lambda_0 \\ 4 & 2-\lambda_2 & -\lambda_1 \\ 4 & 0 & -\lambda_2 \end{pmatrix}.$$

Taking $\lambda_0 = 1, \lambda_1 = 3$ and $\lambda_2 = 2$ cancels the second column and So, will have kernel equal to $Vect(0,1,0)$. Now, Theorem 8(iii) applies to $M - L$:

```
Fractional scaling discrete step
T(x,y) / Q(x,y) = 1 / ((x + y) (x + 2 y))^2
Module of degree 3 and rank 1 and Kernel of dimension 2
{{0, 1, 0}}
Basis of invariant Ideal
{ x y }
```

It was clear from the beginning that the corresponding polynomial x_1x_2 is $1/[(x_1+x_2)(x_1+2x_2)]^2$-scale invariant. For instance it is an invariant for the initial values $(0,1)$. Moreover, it clearly never cancels x_1+x_2 and x_1+2x_2, because they are of the form $(a,0)$ or $(0,b)$ with a and b strictly positive. $\square$

We thus generated a basis of a vectorial space which describes invariants for each location, transitions and initial conditions. A global invariant would be any invariant which is in the intersection of these three vector spaces. In this way, we avoid the definition of a single isomorphism for the whole hybrid system. Instead, we generate the basis for each separate consecution condition. To compute the basis of global invariants, we could use the following theorem. It proposes to *multiply* all the elements of each computed basis. By so doing, we also avoid the heavy computation of ideal intersections This approach is a sound, but not complete, way of computing ideals for global hybrid invariants, and it has a lower computational complexity.

Theorem 9. *Let W be a hybrid system and let l be one of its locations. Let $I = \{I_1, ..., I_k\}$ a set of ideals in $\mathbb{R}[X_1, ..., X_n]$ such that $I_j = ({f^{(j)}}_1, ..., f^{(j)}_{n_j})$*

where $j \in [1,k]$. *Let* $\otimes(I_1,...,I_k) = \{\delta_1,...,\delta_{n_1 n_2 \ldots n_k}\}$ *be such that all elements* δ_i *in* $\otimes(I_1,...,I_k)$ *are formed by the product of one element from each ideal in* I. *Assume that the* I_j*'s are collections of invariant ideals associated to* $\mathcal{S}(l)$, *its differential rule,* $\mathcal{C}(l)$, *its local conditions, and all invariant ideals generated considering all incoming transitions at* l. *Then* $\otimes(I_1,...,I_k)$ *is a non-trivial invariant ideal for location* l.

8 Conclusions

Our methods reduce the non-trivial non-linear invariant generation problem to linear algebra problems. They display lower complexities than the mathematical foundations of the previous approaches that use Grobner basis calculation or quantifier elimination. Our algorithm is capable of computing basis for ideals of non trivial invariants for non linear hybrid systems. It also embodies a strategy to estimate degree bounds which allow for the discovery of rich classes of inductive invariants. Moreover, we provide very general sufficient conditions allowing the existence and computation of invariant ideals. These conditions could be directly used by any constraint-based invariant generation method [6, 5, 9, 8] or by any analysis methods based on over-approximations and reachability [24, 15, 25].

References

[1] Henzinger, T.: The theory of hybrid automata. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996), New Brunswick, New Jersey, pp. 278–292 (1996)

[2] Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2-3), 103–179 (1992)

[3] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)

[4] Manna, Z.: Mathematical Theory of Computation. McGrw-Hill, New York (1974)

[5] Sankaranarayanan, S., Sipma, H., Manna, Z.: Constructing invariants for hybrid system. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 539–554. Springer, Heidelberg (2004)

[6] Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)

[7] Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates (2004)

[8] Tiwari, A.: Generating box invariants. In: Proc. of the 11th Int. Conf. on Hybrid Systems: Computation and Control HSCC (2008)

[9] Sankaranarayanan, S., Dang, T., Ivancic, F.: Symbolic model checking of hybrid systems using template polyhedra. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 188–202. Springer, Heidelberg (2008)

[10] Buchberger, B.: Symbolic computation: Computer algebra and logic. In: Proceedings of the 1st Int. Workshop on Frontiers of Combining Systems, pp. 193–220 (1996)
[11] Weispfenning, V.: Quantifier elimination for real algebra - the quadratic case and beyond. Applicable Algebra in Engineering, Communication and Computing 8(2), 85–101 (1997)
[12] Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. JSAT 1(3-4), 209–236 (2007)
[13] Tiwari, A., Khanna, G.: Nonlinear systems: Approximating reach sets. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 600–614. Springer, Heidelberg (2004)
[14] Rodriguez-Carbonell, E., Tiwari, A.: Generating polynomial invariants for hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 590–605. Springer, Heidelberg (2005)
[15] Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)
[16] Matringe, N., Moura, A.V., Rebiha, R.: Morphisms for non-trivial non-linear invariant generation for algebraic hybrid systems. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 445–449. Springer, Heidelberg (2009)
[17] Matringe, N., Moura, A.V., Rebiha, R.: Morphisms for analysis of hybrid systems. In: ACM/IEEE Cyber-Physical Systems CPSWeek 2009, Second International Workshop on Numerical Software Verification (NSV 2009) Verification of Cyber-Physical Software Systems, San Francisco, CA, USA (2009)
[18] Matringe, N., Moura, A.V., Rebiha, R.: Endomorphisms for non-trivial non-linear loop invariant generation. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 425–439. Springer, Heidelberg (2008)
[19] Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: HSCC 2010: Proc. of the 13th ACM Int. Conf. on Hybrid Systems: Computation and Control, pp. 221–230. ACM, New York (2010)
[20] Matringe, N., Vieira-Moura, A., Rebiha, R.: Morphisms for non-trivial non-linear invariant generation for algebraic hybrid systems. Technical Report TR-IC-08-32, Institute of Computing, University of Campinas (November 2008)
[21] Matringe, N., Vieira-Moura, A., Rebiha, R.: Endomorphism for non-trivial semi-algebraic loop invariant generation. Technical Report TR-IC-08-31, Institute of Computing, University of Campinas (November 2008)
[22] Lang, S.: Algebra. Springer, Heidelberg (January 2002)
[23] Tomlin, C., Pappas, G.J., Sastry, S.: Conflict resolution for air traffic management: a study in multiagent hybrid systems. IEEE Transactions on Automatic Control 43(4), 509–521 (1998)
[24] Piazza, C., Antoniotti, M., Mysore, V., Policriti, A., Winkler, F., Mishra, B.: Algorithmic Algebraic Model Checking I: Challenges from Systems Biology. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 5–19. Springer, Heidelberg (2005)
[25] Ramdani, N., Meslem, N., Candau, Y.: Reachability of uncertain nonlinear systems using a nonlinear hybridization. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 415–428. Springer, Heidelberg (2008)

Linear-Invariant Generation for Probabilistic Programs:
Automated Support for Proof-Based Methods

Joost-Pieter Katoen[1], Annabelle K. McIver[2,⋆],
Larissa A. Meinicke[2,⋆], and Carroll C. Morgan[3,⋆]

[1] Software Modeling and Verification Group, RWTH Aachen University, Germany
[2] Dept. Computer Science, Macquarie University, NSW 2109 Australia
[3] School of Comp. Sci. and Eng., Univ. New South Wales, NSW 2052 Australia

Abstract. We present a constraint-based method for automatically generating *quantitative invariants* for *linear probabilistic programs*, and we show how it can be used, in combination with proof-based methods, to verify properties of probabilistic programs that cannot be analysed using existing automated methods. To our knowledge, this is the first automated method proposed for quantitative-invariant generation.

Keywords: Probabilistic programs, quantitative program logic, verification, invariant generation.

1 Introduction

Verification of sequential programs rests typically on the pioneering work of Floyd, Hoare and Dijkstra [13, 18, 11] in which annotations are associated with control points in the program. For probabilistic programs, quantitative annotations are needed to reason about probabilistic program correctness [25, 8, 27]. We generalise the method of Floyd, Hoare and Dijkstra to probabilistic programs by making the annotations real- rather than Boolean-valued expressions in the program variables [25, 27]. As is well known, the crucial annotations are those used for loops, the loop *invariants*. Thus in particular we focus on real-valued, *quantitative* invariants: they are random variables whose expected value is not decreased by iterations of the loop [29].

One way of finding annotations is to place them speculatively on the program, as parametrised formula containing only first-order unknowns, and then to use a constraint-solver to solve for parameter-instantiations that would make the associated "verification conditions" true [5, 30, 6, 28, 14]. Such approaches are referred to as being *constraint-based*.

Our main contribution in this paper is to generalise the constraint-based method of Colón et al. [5] to probabilistic programs. We demonstrate our generalisation on a number of small-but-intricate probabilistic programs, ones whose

⋆ We acknowledge the support of the Australian Research Council Grant DP0879529.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 390–406, 2010.
© Springer-Verlag Berlin Heidelberg 2010

analyses appear to be beyond other automated techniques for probabilistic programs at this stage. We discuss this in Sec. 7.

We begin in the next section with an overview of our approach.

2 Overall Summary of the Approach

For qualitative (non-probabilistic) programs, Boolean annotations are called *assertions*; and the associated *verification condition* for assertions P and Q separated by a program path *path_prog* (that does not pass through other annotations) is that they must satisfy the Hoare triple [18]

$$\{P\}\ path_prog\ \{Q\} \quad \textit{or equivalently} \quad P \Rightarrow wp.path_prog.Q \ ,$$

where *wp* refers to Dijkstra's *weakest-precondition* semantics of programs [11]. In either formulation, this condition requires that whenever the precondition P holds before the execution of *path_prog*, the postcondition Q holds after.

In the constraint-based method of Colón et al. [5], assertions for *linear programs* –programs with real-valued program variables in which expressions occurring in both conditionals and assignment expressions must be *linear* in the program variables– are found by speculatively annotating a program with Boolean expressions of the particular linear form $a_1x_1 + \ldots + a_nx_n + a_{n+1} \leq 0$, where $a_1, \ldots, a_{n+1}$ are parameters and $x_1, \ldots, x_n$ are program variables. The verification conditions associated with these annotations are then expressed as a set of polynomial constraints over the annotation-parameters and solved (for those unknown parameters) using off-the-shelf SAT solvers. This process yields Boolean annotations, that is assertions, from which program correctness can subsequently be inferred.

For probabilistic programs our real-valued (not Boolean) annotations are called *expectations* (rather than assertions), and the verification condition $\{P\}$ *path_prog* $\{Q\}$ is now interpreted as follows: if *path_prog* takes some initial state σ to a final distribution δ' on states, then the expected value of *post-expectation* Q over δ' is at least the (actual) value of *pre-expectation* P over σ. Using the quantitative *wp* semantics whose definition appears at Fig. 1, this condition is equivalently written as $P \leq wp.path_prog.Q$. When there is no probability, quantitative *wp* is in fact isomorphic to ordinary (qualitative) *wp* [27].

Example 1. Consider a slot machine with three dials and two symbols, hearts ($\heartsuit$) and diamonds ($\diamondsuit$), on each one. The state of the machine is the configuration of the dials: a mapping from dials d_1, d_2 and d_3 to suits. The semantics of program *flip* that spins the dials independently so that they come to rest on each of the suits with equal probability,

$$\mathit{flip} \quad := \quad (d_1 := \heartsuit \ {}_{\frac{1}{2}}\!\oplus d_1 := \diamondsuit); (d_2 := \heartsuit \ {}_{\frac{1}{2}}\!\oplus d_2 := \diamondsuit); (d_3 := \heartsuit \ {}_{\frac{1}{2}}\!\oplus d_3 := \diamondsuit) \ ,$$

is then a function that maps each initial state to a single distribution δ' in which the probability of being in a slot-machine state is $\frac{1}{8}$ for each. If $\mathsf{all}.x$ is

the expression $x{=}d_1{=}d_2{=}d_3$ and $[\cdot]$ is the function that takes false to 0 and true to 1, then we have for example that the expected value of $[\mathsf{all}.\heartsuit]$ over δ', $\mathit{wp.flip}.[\mathsf{all}.\heartsuit] = \frac{1}{8}$, is the probability of reaching final state $\mathsf{all}.\heartsuit$. This means that the probabilistic Hoare triple $\{1/8\}$ *flip* $\{[\mathsf{all}.\heartsuit]\}$ holds.

In general, a post-expectation may be any real-valued expression in the program variables, as the following example shows.

Example 2. Again with *flip*, a post-expectation Q may be used to represent the winnings assigned to each final configuration of suits. For instance, we could have $Q := 1{\times}[\mathsf{all}.\heartsuit] + \frac{1}{2}{\times}[\mathsf{all}.\diamondsuit]$ to represent that a gamer wins the whole jackpot if there are three hearts, a half if there are three diamonds, and nothing otherwise. Pre-expectation $\mathit{wp.flip.Q}$ then represents a mapping from initial configurations of the slot machine to the least fraction of the jackpot the gamer can expect to win from that configuration. For the above Q, we have that $\mathit{wp.flip.Q}$ is a mapping from each state to the value $\frac{3}{16}$, that is $6{\times}\frac{1}{8}{\times}0 + \frac{1}{8}{\times}1 + \frac{1}{8}{\times}\frac{1}{2}$.

Our first main technical contribution is to show how to determine the verification conditions for a probabilistic program annotation. To do this we must identify the appropriate notion of execution paths between annotations: this is because it doesn't make sense to speak of some annotation P's "being true here" when P is a real-valued expression over the program variables (rather than a Boolean predicate). The principal problem is paths through decision points, e.g. conditionals, where the truth (or falsity) of the Boolean condition cannot determine a "dead path" in the way that Colón does: we are not able to formulate a notion of "probably dead." Thus we explain in Sec. 4 below how, by imposing an extra condition on the program annotation, we can avoid this problem. For now we concentrate on the special case of annotating a single loop.

A single loop, $\mathit{loop} := \mathsf{while}\ G\ \mathsf{do}\ \mathit{body}\ \mathsf{od}$, is annotated as follows

$$\{I\};\mathsf{while}\ G\ \mathsf{do}\ \{[G]{\times}I\};\mathit{body}\ \mathsf{od};\{[\neg G]{\times}I\}\ ,$$

where I is some expectation. Such annotations are verifiable (i.e. *valid*) just when the expected value of I does not decrease after an iteration of the loop body, that is

$$[G]{\times}I \quad \leq \quad \mathit{wp.body.I}\ . \tag{1}$$

In this situation we refer to I as a *quantitative invariant* (or invariant) of the loop [29, 27]. In the case that the loop terminates (i.e. it terminates with probability 1), and indeed all of our examples in this paper are terminating, we may reason that if (1) holds so does the probabilistic Hoare triple $\{I\}$ *loop* $\{[\neg G]{\times}I\}$.[1]

Example 3. The behaviour of a gamer that plays the slot-machine described earlier (at least once) until the dials show all hearts or all diamonds is represented by program

$$\begin{array}{ll} \mathit{init}: & \mathit{flip};\\ \mathit{loop}: & \mathsf{while}\ \ \neg(\mathsf{all}.\heartsuit \vee \mathsf{all}.\diamondsuit)\ \mathsf{do}\ \mathit{flip}\ \mathsf{od}\ . \end{array}$$

[1] Quantitative invariants may also be used to reason about loops that terminate with some probability between 0 and 1 (see [27]).

If the potential winnings are again described by Q (from Ex. 2), we can use the invariant $I := \frac{3}{4}\times[\neg(\mathsf{all}.\heartsuit \vee \mathsf{all}.\diamondsuit)] + 1\times[\mathsf{all}.\heartsuit] + \frac{1}{2}\times[\mathsf{all}.\diamondsuit]$ to calculate the gamer's expected winnings. (Playing the machine costs nothing in this simplistic example.) Since I is an invariant of *loop*, which terminates, and Q equals $[\mathsf{all}.\heartsuit \vee \mathsf{all}.\diamondsuit] \times I$, we have that $\{I\}$ *loop* $\{Q\}$ holds. Thus the gamer can expect to win at least $\mathit{wp.init.I} = 6\times\frac{1}{8}\times\frac{3}{4} + \frac{1}{8}\times 1 + \frac{1}{8}\times\frac{1}{2} = \frac{3}{4}$ of the jackpot. (Half the time the loop will terminate showing all hearts and the gamer will win the whole jackpot, and half the time it will terminate with all diamonds and he will win half.)

Our second main technical contribution is to identify in Sec. 5.1 a class of probabilistic programs and parametrised expectations for which machine-solvable verification conditions can readily be extracted. As for Colón et al. [5] the class of probabilistic programs that our method works for is the set of *linear probabilistic programs*: the set of linear qualitative programs that may also contain discrete probabilistic choices made with a constant probability. Using our parametrised expectations it is possible to express invariants like I from Ex. 3.

Our third main technical contribution is to show in Sec. 5.2 how to convert our verification conditions on parametrised annotations to the same form as those generated by Colón et al. [5], so that they can be machine-solved in much the same way. Since this verification-condition translation is an equivalence, our method is both correct and fully general. That is, it can be used to find all parameter solutions that make an annotation valid, and no others.

3 Probabilistic Programs

Probabilistic programs with nondeterministic and discrete probabilistic choices can be written using the probabilistic guarded command language (pGCL); in Fig. 1 we set out its syntax and *wp* semantics. Non-negative real-valued functions that are bounded above by some constant are referred to as *expectations*, and written as expressions in the program variables. For a probabilistic program *prog* and expectation Q, $\mathit{wp.prog.Q}$ represents the *least expected value* of Q in the final-state of *prog* (as an expression on the initial value of the program variables). This semantics is dual to an operational-style interpretation of program execution, where from an initial state σ the result of a computation is a set of probability distributions over final states; it is dual because $\mathit{wp.prog.Q}$ evaluated at σ is exactly the minimal expected value of Q over any of the result distributions. When Q is of the form $[R]$ for some Boolean expression R, then *wp* is in fact just the least probability that the final state will satisfy R, as in Example 1 above; but it can be more generally applied, as in Example 2.

Probabilistic guarded commands are *scaling*, $c{*}\mathit{wp.prog.Q}{=}\mathit{wp.prog.}(c{*}Q)$, and *monotonic*, $Q_1{\le}Q_2{\Rightarrow}\mathit{wp.prog.}Q_1{\le}\mathit{wp.prog.}Q_2$, for all expectations Q, Q_1, Q_2 and constants c [27]. Scaling, for example, is essentially linearity of expected values.

4 Probabilistic Program Annotations

If we imagine a program as a flowchart, a program annotation associates predicates with arcs and conventionally has the interpretation that a predicate is true

	prog	$wp.prog.Q$
Identity	skip	Q
Assignment	$x := E$	$Q[x\backslash E]$
Composition	$prog_1; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Nondet. choice	$prog_1 \sqcap prog_2$	$wp.prog_1.Q \sqcap wp.prog_2.Q$
Prob. choice	$prog_1 \ {}_p\!\oplus\ prog_2$	$p{*}wp.prog_1.Q + (1{-}p){*}wp.prog_2.Q$
While-loop	while G do *body* od	$(\mu X \cdot [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is an expression in the program variables; $prog_{\{1,2\}}$ and *body* are probabilistic programs; G is a Boolean-valued expression in the program variables; p is a constant probability in $[0, 1]$; and Q is an expectation (represented as a real-valued expression in the program variables). We write $Q[x\backslash E]$ to mean expression Q in which free occurrences of x have been replaced by expression E.

For expectations (interpreted as real-valued functions), scalar multiplication $*$, multiplication, $\times$, addition, $+$, subtraction, $-$, minimum, $\sqcap$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual pointwise extension of these operators (as they apply to the real numbers). Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, minimum and finally the comparison operators. Operators of equal precedence are evaluated from the left. μ is the least fixed point operator w.r.t. the ordering $\leq$ between expectations.

Function $[\cdot]$ takes Boolean expression *false* to 0 and *true* to 1. For $\{0, 1\}$-valued functions, operation $\leq$ has the same meaning as implication over predicates, and $\times$ and $\sqcap$ represent conjunction, and addition over disjoint predicates is equivalent to disjunction.

Fig. 1. Probabilistic program notation and weakest-precondition semantics

of the program state whenever its associated arc is traversed during execution. Our generalisation of qualitative program annotations is to replace predicates (Boolean-valued expressions over the program variables) with expectations.

In order to specify verification conditions on these annotations, we impose restrictions on the program annotations that we allow: first, as for the qualitative case, there must be at least one annotation along any cyclic program path. This is so that verification conditions only involve cycle-free program fragments. Second, we assume that there is an annotation at the beginning and end of the program so that we can reason about the correctness of the whole. The third restriction is made so that we can reason about the branching behaviour of probabilistic programs. We require that if there is any "interior" annotation on a while-loop, conditional, nondeterministic or probabilistic choice, i.e. one following its choice point but occurring before the two choices rejoin, then the choice point itself must have three "immediate" annotations as well: one at its entry, and one at each of its (two) exits. Thus if we consider the flowchart generated by the annotated program fragment

$$\{P\}; prog_1; \text{if } G \text{ then } prog_2; \{Q\}; prog_3 \text{ else } prog_4 \text{ fi}; \{R\}\ , \tag{2}$$

we see that the conditional "if G" has an interior annotation Q — and so we must augment (2) with further annotations S, T, U as follows:

$$\{P\}; prog_1; \{S\}; \mathsf{if}\ G\ \mathsf{then}\ \{T\}; prog_2; \{Q\}; prog_3\ \mathsf{else}\ \{U\}; prog_4\ \mathsf{fi}; \{R\}\ , \quad (3)$$

The S annotation is just before the choice point "if G"; the T annotation is just after its *true* exit; and the U annotation is just after its (implied) *false* exit. Loops and the other kinds of choice statements are similarly annotated.

A program annotation is *valid* when it satisfies the following verification conditions:

- For every pair (P, Q) of annotations separated by a program *path_prog* that does not contain annotations, if P does not appear just before a choice-point with an interior annotation then $\{P\}$ *path_prog* $\{Q\}$ holds. For example, for (3) we must have that $\{P\}$ $prog_1$ $\{S\}$, $\{T\}$ $prog_2$ $\{Q\}$, $\{Q\}$ $prog_3$ $\{R\}$ and $\{U\}$ $prog_4$ $\{R\}$ hold.
- Annotations appearing just before choice-points with interior annotations must be treated differently. In the case of program (3) for instance, it makes no sense to give a meaning to the Hoare triple $\{S\}$ "G is *true*" $\{T\}$. For annotation S in (3) we require that the "special" constraint $S \leq [G]\times T + [\neg G]\times U$ –that does not involve program execution at all– holds. Choice-point annotations on nondeterministic and probabilistic choices and while-loops must satisfy similar constraints. For example, annotation P in fragment $\{P\}; (\{Q\}; prog_1 \sqcap \{R\}; prog_2)$ must satisfy $P \leq Q \sqcap R$. Likewise, for $\{P\}; (\{Q\}; prog_1\ {}_p\!\oplus \{R\}; prog_2)$ we must have $P \leq p{*}Q + (1{-}p){*}R$.

Theorem 1. *Given a valid annotation of a terminating probabilistic program prog such that the first annotation is P and the last is Q, we have that prog satisfies the probabilistic Hoare triple $\{P\}$ prog $\{Q\}$.*

Proof. By structural induction over program texts.

For example, if (3) terminates and the annotation is valid then the probabilistic Hoare triple $\{P\}\ prog_1; \mathsf{if}\ G\ \mathsf{then}\ prog_2; prog_3\ \mathsf{else}\ prog_4\ \mathsf{fi}\ \{R\}$ holds.

4.1 The Special Case of Loops

In this paper we deal only with single-loop programs (mostly) and the conditions above require that a loop be annotated (at least) with an expectation just before the loop, one just before the loop body (the *true* branch of the loop conditional) and one just after the loop (the *false* branch). For $loop := \mathsf{while}\ G\ \mathsf{do}\ body\ \mathsf{od}$, this amounts to the following annotation, $\{I\}; \mathsf{while}\ G\ \mathsf{do}\ \{J\}; body\ \mathsf{od}; \{K\}$, which is valid if

$$\{J\}\ body\ \{I\} \quad \textit{and special constraint} \quad I \leq [G]\times J + [\neg G]\times K$$

holds. We simplify this further by taking J to be $[G]\times I$ and K to be $[\neg G]\times I$ so that the special constraint is satisfied by construction — we need only find an I so that $\{[G]\times I\}$ *body* $\{I\}$. Such an I is referred to as a *quantitative invariant.*

5 Constraint-Solving for Quantitative Annotations

Given a "linear probabilistic program" annotated with parametrised real-valued expressions that are "propositionally linear" (the definitions for which appear in the following section), we show how to extract a set of polynomial constraints that are sufficient and necessary to show that the annotation is valid. Once we have the constraints we are able to apply constraint solvers to solve for the annotation parameters.

5.1 Linear Probabilistic Programs and Parametrised Annotations

An expression E on a given state space is *linear* if it is a linear combination of the program variables. A predicate P is a *linear constraint* if it is an inequality *or* a strict inequality between linear expressions. A *linear assertion* is then a finite conjunction of linear constraints. Finally, for any natural-valued constants M and N, and linear constraints P_{mn}, Boolean expression $(\bigwedge_{m:\,[1..M]} \bigvee_{n:\,[1..N]} P_{mn})$ is said to be a *propositionally linear predicate* with *conjunctive-degree* M and *disjunctive-degree* N.

A quantitative expression of the form $\sum_{m:\,[1..M]}[\bigwedge_{n:\,[1..N]} P_{mn}] \times Q_m$, where M and N are naturals, each P_{mn} is a linear constraint and Q_m is a linear expression, is referred to as a *propositionally linear expression* with *additive-degree* M and *conjunctive-degree* N. Such an expression is written in *disjoint normal form* if for all $i, j\colon [1..M]$ where $i \neq j$, we have $(\bigwedge_{n:\,[1..N]} P_{in}) \wedge (\bigwedge_{n:\,[1..N]} P_{jn}) = \mathit{false}$.

Lemma 1. *Any propositionally linear expression is semantically equivalent to another propositionally linear expression in disjoint normal form. (See App. C for proof.)*

A probabilistic program is said to be *linear* if the variables are real-valued, all of the guards are linear constraints, and updates are linear expressions.

To find valid quantitative annotations for a program with variables $x_1, \ldots, x_X$, we parametrise each annotation with a propositionally linear expression

$$\begin{gathered}\textstyle\sum_{m:\,[1..M]}[\bigwedge_{n:\,[1..N]} \alpha_{(j,mn,1)}x_1 + \ldots + \alpha_{(j,mn,X)}x_X + \beta_{(j,mn)} \ll 0] \\ \times\, (\gamma_{(j,m,1)}x_1 + \ldots + \gamma_{(j,m,X)}x_X + \delta_{(j,m)})\end{gathered}$$

containing free real-valued variables $\alpha_{(j,mn,x)}$, $\beta_{(j,mn)}$, $\gamma_{(j,m,x)}$ and $\delta_{(j,m)}$, in which each occurrence of $\ll$ is instantiated to either $<$ or $\leq$.

To ensure that each annotation P is an expectation (a real-valued expression bounded below by 0 and above by some real number) we require $0 \leq P \leq 1$. Restricting each annotation to be bounded above by 1 instead of an arbitrary upper bound does not limit our method because programs satisfy scaling (Sec. 3) so that if *prog* is correctly annotated with expectations P, Q, etc., then the modified annotation in which P is replaced by $c{*}P$, Q is replaced by $c{*}Q$ etc. is also valid for any non-negative constant c.

5.2 Constructing Machine-Solvable Constraints

For qualitative programs the verification conditions for a linear program annotated with linear constraints can be formulated as Boolean expressions on linear constraints. After rewriting these expressions in conjunctive normal form (as propositionally linear predicates), Colón et al. [5] showed how it was possible to use Motzkin's Transposition theorem [15] to reduce each (universally quantified) finite disjunction of linear constraints to an existentially quantified polynomial formula over the annotation parameters.[2]

For probabilistic programs, the verification conditions for a program annotation are of one of the five possible forms (Sec. 4):

$$0 \leq P \leq 1 \tag{4}$$

$$P \leq \mathit{wp.path_prog.Q} \tag{5}$$

$$R \leq [G]{\times}S + [\neg G]{\times}T \quad \mathit{or} \quad R \leq S \sqcap T \quad \mathit{or} \quad R \leq p{*}S + (1{-}p){*}T \tag{6}$$

where P, Q, R, S and T are annotations, G is a Boolean expression in the program variables, p is a constant in $[0, 1]$, and *path_prog* is a loop- and annotation-free program fragment occurring in the program. For linear probabilistic programs, G must be a linear constraint, and sub-program *path_prog* must also be linear. By parametrisation the annotations are propositionally linear.

To convert each constraint of the form (4–6) to machine-solvable form we must first formulate each of them as a finite Boolean expression on linear constraints. We can then use Colón et al.'s method to convert these to polynomial formulae over the annotation parameters. We start by showing how this novel first step is performed and then we recount Motzkin's transposition theorem.

Equivalence translation to a finite Boolean expression. This translation occurs in two stages. First we convert each expression of the form (4–6) to inequalities between propositionally linear expressions. This first step is made possible by the following observations:

Lemma 2. *Let S and T be any propositionally linear expressions, G a linear constraint, p a constant expression, x a program variable, and E a linear expression. We have that $[G] \times S$, $S \sqcap T$ and $p{*}S$ are semantically equivalent to propositionally linear expressions; $\neg G$ may be expressed as a linear constraint; $S + T$ and $S[x \backslash E]$ are propositionally linear. (See App. C for proof.)*

Using these observations we immediately have that constraints of the form (4) and (6) can be translated to inequalities between propositionally linear expressions. For (5), we may use them to show by structural induction that *wp.path_prog.Q* may be evaluated to a propositionally linear expression.

[2] To be precise, Colón et al. [5] used a specialisation of this theorem, Farkas' lemma, since they did not consider parametrised forms of invariants that could include strict inequalities.

In the second step we convert each of these inequalities between propositionally linear expressions to finite Boolean expressions over linear constraints (which may then be translated to conjunctive normal form):

Lemma 3. *Any inequality $Q_a \leq Q_b$ between non-negative propositionally linear expressions can be equivalently formulated as a finite Boolean expression over linear constraints.*

Proof. First rewrite Q_a and Q_b in disjoint normal form (Lem. 1) as propositionally linear expressions $[P_{a1}]\times Q_{a1} + \cdots + [P_{aM}]\times Q_{aM}$, and $[P_{b1}]\times Q_{b1} + \cdots + [P_{bK}]\times Q_{bK}$, where each P_{am}, P_{bk} are linear assertions and Q_{am}, Q_{bk} are linear expressions. Then $Q_a \leq Q_b$ if and only if for all $m\colon [1..M]$ and $k\colon [1..K]$ we have $P_{am} \wedge P_{bk} \Rightarrow (Q_{am} - Q_{bk} \leq 0)$ and $P_{am} \wedge (\bigwedge_{k\colon [1..K]} \neg P_{bk}) \Rightarrow (Q_{am} \leq 0)$.

Equivalence translation using Motzkin's transposition theorem. Motzkin's Transposition theorem can be used to equivalently represent any (universally quantified) propositionally linear predicate as a conjunction of existentially quantified constraints. Since the linear constraints in our propositionally linear predicate contain unknown coefficients, the constraints derived from Motzkin's Transposition theorem are polynomial, and not linear.

Theorem 2. *Motzkin's Transposition Theorem* Given the set of linear, and strict linear, inequalities over real-valued variables $x_1, ..., x_n$

$$S := \begin{bmatrix} \alpha_{(1,1)}x_1 & + \ldots + & \alpha_{(1,n)}x_n & + \beta_1 & \leq 0 \\ \vdots & & \vdots & \vdots & \\ \alpha_{(m,1)}x_1 & + \ldots + & \alpha_{(m,n)}x_n & + \beta_m & \leq 0 \end{bmatrix}$$

$$T := \begin{bmatrix} \alpha_{(m+1,1)}x_1 & + \ldots + & \alpha_{(m+1,n)}x_n & + \beta_{m+1} & < 0 \\ \vdots & & \vdots & \vdots & \\ \alpha_{(m+k,1)}x_1 & + \ldots + & \alpha_{(m+k,n)}x_n & + \beta_{m+k} & < 0 \end{bmatrix} ,$$

in which $\alpha_{(1,1)}, ..., \alpha_{(m+k,n)}$ and $\beta_1, ..., \beta_{m+k}$ are real-valued, we have that S and T simultaneously are *not* satisfiable (i.e. they have no solution in x) if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_{m+k}$ such that either

$$0 = \textstyle\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \quad \ldots \quad , 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, \ \ 1 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0 \ ,$$

or at least one coefficient λ_i for i in the range $[m+1 \ldots m+k]$ is non-zero and

$$0 = \textstyle\sum_{i=1}^{m+k} \lambda_i \alpha_{(i,1)}, \quad \ldots \quad , 0 = \sum_{i=1}^{m+k} \lambda_i \alpha_{(i,n)}, \ \ 0 = (\sum_{i=1}^{m+k} \lambda_i \beta_i) - \lambda_0 \ .$$

Proof. This is a geometric rephrasing of the theorem as it appears in a standard reference [15, p.268].

5.3 Solving Constraints and Heuristics

Constraint solving. Our generated constraints are of the same form as those generated by Colón et al. [5] for qualitative programs, and may therefore be solved using exactly the same tools and techniques applicable there.

A survey of techniques for solving constraints is given by Bockmayr and Weispfenning [2]. Colón et al. [5] used, for example, REDLOG's[3] [12] implementation of quantifier-elimination algorithms for polynomial constraints. In addition to quantifier-elimination techniques, other methods such as factorisation and root finding were employed.

Quantifier-elimination implementations are exponential in complexity, which limits the size of annotation-generation problems that may be addressed using this approach. In our examples from Sec. 6 –which are of a small size– we solved our constraints using REDLOG.

Heuristics. In practice it may not be possible automatically to solve constraints when the program size is large or the parametrised invariants have either additive- or conjunctive- degree greater than say two or three or, even if we can, still the output of quantifier-elimination procedures might be unreadable. Colón et al. [5] encountered similar problems for trying to generate "k-linear inductive assertions" for values of k greater than one. As in [5] we recommend, where possible, (i) reducing the size of a problem by guessing values of certain parameters, and (ii) decomposing the task into finding structurally smaller invariants and (iii) finding invariants for sub-programs separately. Other suggestions (such as polynomial factorisation) may be found in [5].

To illustrate (ii), we have for instance that for linear assertion P and propositionally linear expression J the expression $I := [P]\times J$ is an invariant of the loop $\mathsf{while}\ G\ \mathsf{do}\ body\ \mathsf{od}$ if $[P]$ is invariant and $0 \leq I \leq 1$ and $[G]\times I \leq wp.body.J$ holds. This method of decomposing the problem –although often applicable– is not complete. That is, there exist loop invariants of the form $[P]\times J$, where P is a linear assertion and J is a linear expression, such that $[P]$ on its own is not invariant.

Example 4. Consider program $x, y := 1, 1; \mathsf{while}\ y < N\ \mathsf{do}\ (y := 2y\ {}_{1/2}\oplus\ x := 0)\ \mathsf{od}$, in which N is a positive constant. Although $[x = 1 \wedge 0 \leq y \leq 2N]$ isn't an invariant of the loop since x is not guaranteed to remain at the value 1, $[x = 1 \wedge 0 \leq y \leq 2N]\times y$ is, since transitions that set x to 0 are balanced by y's doubling in value.

5.4 Soundness and Completeness

Theorem 3. *For any linear probabilistic program annotated with propositionally linear expressions, our method is correct and fully general. That is, it can be used to find all parameter solutions that make the annotation valid, and no others.*

[3] Available from http://redlog.dolzmann.de/.

$$\begin{array}{ll} \textit{init}: & x, n := 0, 0; \\ \textit{loop}: & \textsf{while } n < N \textsf{ do} \\ \textit{body}: & \quad (x := x + 1 \ {}_p\oplus \textsf{skip}); n := n + 1 \\ & \textsf{od} \end{array}$$

Variables x and n are of type $\mathbb{N}$, constant $N\colon \mathbb{N}$, and constant $p\colon [0,1]$. To verify that the expected final value of x is at least $p{\times}N$ we must show that $wp.(init; loop).x \geq pN$, which is implied by the discovered loop invariant $[0{\leq}x{\leq}\, n \wedge n{\leq}N] \times (\alpha x - p\alpha n + p\alpha N)$, where $0 \leq \alpha \leq 1/N$.

Fig. 2. Binomial update

Proof. This follows from the fact that our translation of the annotation verification conditions to machine-solvable form (as defined in Sec. 4) is an equivalence.

This means, for instance, that our method can be used to find all propositionally linear invariants of a chosen degree for a single (i.e. un-nested) loop.

6 Three Examples

We will now use the invariant-generation method set out on the preceding sections together with proof-based techniques to analyse three simple, terminating[4], probabilistic programs. The Boolean and natural-valued variables are interpreted more generally as reals (with Boolean value *true* represented by real-value 1 and *false* by 0), so that we can apply our approach.

6.1 Example One: Binomial Update

The program in Fig. 2 sets variable x to a value between 0 and constant N according to the binomial distribution with parameter p. We use our invariant-generation method to find invariants of *loop* for calculating lower-bounds on the final expected value of x.

We first search for invariants for *loop* of the form $I := [\alpha x + \beta n + \gamma \leq 0]$, that we can use to describe upper and lower bounds on the values of program variables x and n. In other words, we search for parameters α, β and γ that make the following program annotation valid:

$$\{I\}; \textsf{while } n{<}N \textsf{ do } \{[n{<}N] \times I\}; (x := x{+}1 \, {}_p{\oplus} \textsf{skip}); n := n{+}1 \textsf{ od}; \{[n{\geq}N] \times I\} \;.$$

Solving for the constraints on the parameters, we find that $[0 \leq x]$, $[x \leq n]$ and $[n \leq N+1]$ are invariants.[5] Next, we search for quantitative invariants for *loop*

[4] In each case a separate (and very simple) argument can be used to show that the programs terminate with probability 1.

[5] Invariant $[n \leq N]$ *cannot* be generated since –although n only takes natural values in the context of the program– we are solving constraints over the reals, and not the natural numbers.

of the form $I := J\times(\alpha x + \beta n + \gamma)$. where $J := [0 \le x \wedge x \le n \wedge n \le N]$. Since J is invariant it suffices to show that for all values of x and n we have

$$0 \le I \quad and \quad I \le 1 \quad and \quad [n < N]\times(\alpha x+\beta n+\gamma) \le wp.body.(\alpha x+\beta n+\gamma) \quad (7)$$

where $wp.body.(\alpha x+\beta n+\gamma)$ can be evaluated to $\alpha x+\beta n+p\alpha+\beta+\gamma$. Using the result of this wp-calculation, constraints (7) may then be equivalently formulated as the following finite Boolean expressions on linear constraints:

$$0 \le x \wedge x \le n \wedge n \le N \;\Rightarrow\; (0 \le \alpha x + \beta n + \gamma) \quad (8)$$
$$0 \le x \wedge x \le n \wedge n \le N \;\Rightarrow\; (\alpha x + \beta n + \gamma \le 1) \quad (9)$$
$$n < N \;\Rightarrow\; (0 \le p\alpha + \beta)\ . \quad (10)$$

To translate (8), (9) and (10) into a set of existentially quantified constraints that can be used as inputs to a SAT-solver, we use Motzkin's Theorem. Condition (10) for instance, which holds if the strict linear inequalities

$$\begin{bmatrix} 0x + \;\; n + \quad\;\; -N < 0 \\ 0x + 0n + p\alpha + \beta < 0 \end{bmatrix}$$

are *not* satisfiable, is equivalent (by Motzkin's Theorem) to the following polynomial constraints:

$$\exists \lambda_0, \lambda_1, \lambda_2 \cdot$$
$$\begin{pmatrix} \lambda_0 \ge 0 \wedge \lambda_1 \ge 0 \wedge \lambda_2 \ge 0 \wedge \\ 0 = \lambda_1 0 + \lambda_2 0 \wedge \\ 0 = \lambda_1 + \lambda_2 0 \wedge \\ 1 = -\lambda_1 N + \lambda_2(p\alpha+\beta) - \lambda_0 \end{pmatrix} \vee \begin{pmatrix} \lambda_0 \ge 0 \wedge \lambda_1 \ge 0 \wedge \lambda_2 \ge 0 \wedge \\ (\lambda_1 \ne 0 \vee \lambda_2 \ne 0) \wedge \\ 0 = \lambda_1 0 + \lambda_2 0 \wedge \\ 0 = \lambda_1 + \lambda_2 0 \wedge \\ 0 = -\lambda_1 N + \lambda_2(p\alpha+\beta) - \lambda_0 \end{pmatrix}$$

Simplifying this constraint reveals that parameters α, β and γ must satisfy $p\alpha + \beta \ge 0$. This condition is satisfied, for example, if $\beta = -p\alpha$ and $\gamma = p\alpha$. Assuming that $\beta = -p\alpha$ and $\gamma = p\alpha$ holds, we have that (8) and (9) hold if N is positive and $0 < \alpha \le 1/N$. Consequently, for positive N and $\alpha\colon (0, 1/N]$ we have that

$$J\times(\alpha x - p\alpha n + p\alpha N) \quad (11)$$

is invariant. Assuming N is positive, (11) can be used to calculate a lower bound on the expected value of x produced by the binomial program. We have

$$\begin{array}{lll} & wp.(init; loop).(\alpha x) & \\ \ge & wp.(init; loop).([n \ge N]\times J\times\alpha x) & \text{“}\alpha x \ge [n \ge N]\times J\times\alpha x\text{; monotonicity”} \\ = & wp.init.(wp.loop.([n \ge N] \times I)) & \text{“simplify; sequential composition”} \\ \ge & wp.init.I & \text{“}loop\text{ terminates and }I\text{ is invariant; monotonicity”} \\ = & [0 \le N] \times p\alpha N & \text{“calculate”} \\ = & p\alpha N\ . & \text{“we have assumed that }N\text{ is positive”} \end{array}$$

From scaling (Sec. 3), a lower bound of the least expected value of x (i.e. $(1/\alpha)\times\alpha x$) that may be produced by the Binomial program is pN (i.e. $(1/\alpha)\times p\alpha N$).

```
init : x := p; b := true;
loop : while b do
           b := false 1/2⊕ true;
           if b then
               x := 2x; if x ≥ 1 then x := x−1 else skip fi
           elseif x ≥ 1/2 then x := 1
           else  x := 0  fi
       od
```

Variable x is of type $\mathbb{R}$ and b of type $\mathbb{B}$. This program sets x to 1 with probability at least p, a fact verified by establishing $wp.(init; loop).[x = 1] \geq p$, which follows from the discovery of the loop invariant $[0 \leq x \leq 1] \times x$.

Fig. 3. Generating a biased coin from a fair one

6.2 Example Two: Generating a Biased Coin from a Fair One

The program in Fig. 3 (which appears in [19, Ch4]) uses a stream of fair coin flips to generate a (single) biased coin. To verify that on termination it correctly sets x to 1 with probability (at least) p, we need to determine that $p \leq wp.loop.[x = 1]$. We used our techniques to discover that $[0 \leq x \leq 1] \times x$ is an invariant of *loop*, and then additional reasoning to show that it implies correctness. First, it simplifies to the post-expectation on termination (because on exiting the loop x takes only the values 0 or 1); next substituting values for the initialisation $x := p$ yields the required lower bound.[6] Details of the generating constraints are set out in App. A.

6.3 Example Three: Uniform Distribution; Nested Loops

Cryptographic applications often require a variable to be chosen uniformly from some interval $[0 \ldots N]$; in practice this must be achieved using a fair coin as above, and the program in Fig. 4 is an example. Intuitively its inner loop sets g uniformly to some interval $[0 \ldots c]$ where c is the smallest power of 2 exceeding N (i.e. $2^{\lceil \log_2 N \rceil}$); the function of the outer loop is to repeat the process from scratch until g lies in the required interval $[0 \ldots N]$.

To verify this program we use the above technique to generate automatically a linear invariant for the inner loop. We then use that invariant to reason *manually* about the effect of the outer loop. In the conclusion we suggest ways in which we might be able to extend our method so that this (manual) reasoning could be automated as well.

Verifying this program thus requires the combination of automated invariant generation and interactive proof, and in this section we sketch how it was done.

Interactive proof: First, we make an assumption that the inner loop correctly sets g uniformly within $[0 \ldots c]$; this is formalised by the set of (parametrised) Hoare triples

$$\{1/c\} \quad init_2; loop_2 \quad \{[g = k]\}\ , \quad 0 \leq k \leq c\ . \tag{12}$$

[6] It can be in fact be shown that this bound on the expected value of x is tight.

$init_1$: $n, g := 1, N$;
$loop_1$: while $g \geq N$ do
$init_2$: $n, g := 1, 0$;
$loop_2$: while $n < N$ do $n := 2n$; $(g := 2g \; _{1/2}\oplus \; g := 2g + 1)$ od
od

n and g are both variables of type $\mathbb{N}$ and N is a constant positive natural number. This program uniformly sets g to a value in $[0..N)$, verified by establishing $wp.init_1; loop_1.[g = k] \geq 1/N$, implied by the invariant (for the outer loop) $[g = k]+[g \geq N] \times (1/N)$.

Fig. 4. Uniform distribution

With this assumption we are able to use the wp-calculus directly to verify that $[g = k] + [g \geq N] \times d$ (for $d \leq 1/N$) is an invariant of the outer loop, $loop_1$, and that is sufficient to verify the whole program; the details are set out at App. B. That leaves us with the problem of establishing (12) ; we use our automated invariant generation technique to find invariants to do so.

Automatic invariant generation for the inner-loop analysis: First we considered the special case of (12) where $k = 0$ and searched for $loop_2$ invariants of the form $I := [g = 0 \wedge J] \times (\alpha n + \gamma)$, where $J := 1 \leq n \leq c$, and we needed that $[J]$ is an invariant of $loop_2$. This gave us $[g = 0 \wedge 1 \leq n \leq c] \times n/c$ as an invariant, which is sufficient for the special case. To generalise this, we then searched for $loop_2$ invariants of the form $[\alpha n + \beta < g \leq \gamma n + \delta \wedge J] \times dn$, and we found that for any α and $dc = 1$, $[\alpha n - 1 < g \leq \alpha n \wedge J] \times dn$ is also invariant, from which we can derive our result. Details are set out in App. B.

7 Alternative Automated Methods

Markov decision processes (MDP's) are a natural candidate for an operational model for probabilistic programs. Analysis of quantitative properties relative to Markov decision processes are available via probabilistic model checking. Examples include PRISM [17] (supporting PCTL model checking) and LiQuor [4] (supporting LTL). Whilst the invariant technique produces general statements about program behaviour, model checking is restricted to the verification of particular instances: for the generation of a biased coin from a fair coin, it can be checked whether eventually the probability that x equals 1 is p, for a given p. One cannot check that for any p this property holds.

Recent developments using abstraction refinement increase the potential for generality. In particular PASS [16] and a SAT-based extension of PRISM [24] both compute sound approximations of the underlying MDP, with the former yielding over approximations, and the latter computing both upper- and lower bounds. In neither case does it appear that the methods could feasibly be used to treat the examples in this paper, however. In particular the analysis of loops by the extension of PRISM tends to be extremely costly, and do not perform well when the variables can take real values [23].

Testing for language equivalence between probabilistic programs over finite integer datatypes has been exploited by the tool APEX [26], but again this would not be able to treat the examples that use real-valued variables straightforwardly.

Abstract interpretation methods [7] have also been applied to probabilistic programs [9, 10]. As for non-probabilistic abstract interpretation methods, these might –in contrast to constraint-based methods– only produce "approximate answers".

Finally, none of PASS, SAT-based PRISM, APEX nor the probabilistic abstract interpretation methods generate quantitative loop invariants.

8 Aims and Conclusions

We have defined a constraint-based method for generating propositionally linear annotations for linear probabilistic programs, and demonstrated it using a number of realistic (but small) probabilistic programs. We have primarily focused on generating invariants for loops. As for other constraint-based methods, the program-size and the size of the parametrised invariants is constrained to small-to-medium sized problem instances by the capabilities of current constraint-solving tools.

Once found, quantitative invariants can be used to prove very general properties of probabilistic programs. Practical experience in automating proofs in HOL [20, 3] has shown that some of the quantitative invariants crucial to proof are not at all obvious; the development of an automated assistant for invariant discovery to augment interactive proofs is one of the main motivations for this work. Our third example in Sec. 6.3 is typical of how invariant generation can enhance an interactive proof session. It also suggests that propositionally linear annotations are unlikely to be sufficient *in themselves* for proving all properties of interest: recall that we used a *set* of discovered linear annotations to approximate the inner loop behaviour. On the other hand, this suggests a method in which sets of annotation pairs could be used more generally to abstract from program behaviour. For us that implies the following hierarchical method: first linear invariants are discovered for inner loops, and then used to abstract the loops' behaviour as sets of annotations. The analysis of all the enclosing loop(s) can then proceed as outlined in this paper, but with the inner loops summarised by their sets of annotations. That is our next step.

Beyond that, we would also like to build tool support for our approach. This would involve, among other tasks, the mechanisation of weakest-precondition calculations involving propositionally linear expressions over probabilistic programs. Earlier mechanisations of the quantitative logic for pGCL (e.g. [20]) suggest that this task is feasible.

Finally, it would be interesting to consider whether other advances in constraint-based invariant generation methods, such as [31, 6, 21] could be adapted to generate polynomial forms of quantitative invariants.

<u>THE APPENDICES</u> A, B and C to this paper may be found online [22].

References

[1] Probabilistic Systems Group, http://www.cse.unsw.edu.au/~carrollm/probs
[2] Bockmayr, A., Weispfenning, V.: Solving numerical constraints. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, ch.12. vol. I, pp. 751–842. Elsevier Science, Amsterdam (2001)
[3] Celiku, O.: Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs. PhD thesis, TUCS (2006)
[4] Ciesinski, F., Baier, C.: LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: Quantitative Evaluation of Systems (QEST), pp. 131–132. IEEE Computer Society Press, Los Alamitos (2006)
[5] Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
[6] Cousot, P.: Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)
[7] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (PoPL), pp. 238–252. ACM, New York (1977)
[8] den Hartog, J., de Vink, E.P.: Verifying probabilistic programs using a Hoare like logic. Int. J. Found. Comput. Sci. 13(3), 315–340 (2002)
[9] Di Pierro, A., Wiklicky, H.: Concurrent constraint programming: towards probabilistic abstract interpretation. In: Gabbrielli, M., Pfenning, F. (eds.) Principles and Practice of Declarative Programming (PPDP), pp. 127–138. ACM, New York (2000)
[10] Di Pierro, A., Wiklicky, H.: Measuring the precision of abstract interpretations. In: Lau, K. (ed.) LOPSTR 2000. LNCS, vol. 2042, pp. 147–164. Springer, Heidelberg (2001)
[11] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
[12] Dolzmann, A., Sturm, T.: REDLOG: computer algebra meets computer logic. SIGSAM Bull. 31(2), 2–9 (1997)
[13] Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science. Proc. Symp. Appl. Math., vol. 19, pp. 19–32. American Mathematical Society, Providence (1967)
[14] Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. Programming Language Design and Implementation (PLDI) 43(6), 281–292 (2008)
[15] Hazewinkel, M.: Encyclopedia of Mathematics. Springer, Heidelberg (2002)
[16] Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
[17] Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
[18] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
[19] Hurd, J.: Formal Verification of Probabilistic Algorithms. PhD thesis, University of Cambridge (2002)
[20] Hurd, J., McIver, A.K., Morgan, C.C.: Probabilistic guarded commands mechanised in HOL. Theoretical Computer Science 346(1), 96–112 (2005)

[21] Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Deduction and Applications (2005)
[22] Katoen, J.P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-invariant generation for probabilistic programs: automated support for proof-based methods. Draft of this paper including its appendices [1, Katoen:10] (2010)
[23] Kattenbelt, M.: Private communication (2010)
[24] Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Abstraction refinement for probabilistic software. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
[25] Kozen, D.: Semantics of probabilistic programs. Jnl. Comp. Sys. Sciences 22, 328–350 (1981)
[26] Legay, A., Murawski, A.S., Ouaknine, J., Worrell, J.: On automated verification of probabilistic programs. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008)
[27] McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, Heidelberg (2004)
[28] Monniaux, D.: Abstract interpretation of probabilistic semantics. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 322–339. Springer, Heidelberg (2000)
[29] Morgan, C.C.: Proof rules for probabilistic loops. In: Jifeng, H., Cooke, J., Wallis, P. (eds.) BCS-FACS 7th Refinement Workshop, Workshops in Computing. Springer, Heidelberg (1996)
[30] Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
[31] Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: Principles of Programming Languages (PoPL), pp. 318–329. ACM, New York (2004)

Abstract Interpreters for Free

Matthew Might

University of Utah, Salt Lake City, Utah, USA
might@cs.utah.edu
http://matt.might.net/

Abstract. In small-step abstract interpretations, the concrete and abstract semantics bear an uncanny resemblance. In this work, we present an analysis-design methodology that both explains and exploits that resemblance. Specifically, we present a two-step method to convert a small-step concrete semantics into a family of sound, computable abstract interpretations. The first step re-factors the concrete state-space to eliminate recursive structure; this refactoring of the state-space simultaneously determines a store-passing-style transformation on the underlying concrete semantics. The second step uses inference rules to generate an abstract state-space and a Galois connection simultaneously. The Galois connection allows the calculation of the "optimal" abstract interpretation. The two-step process is unambiguous, but nondeterministic: at each step, analysis designers face choices. Some of these choices ultimately influence properties such as flow-, field- and context-sensitivity. Thus, under the method, we can give the emergence of these properties a graph-theoretic characterization. To illustrate the method, we systematically abstract the continuation-passing style lambda calculus to arrive at two distinct families of analyses. The first is the well-known k-CFA family of analyses. The second consists of novel "environment-centric" abstract interpretations, none of which appear in the literature on static analysis of higher-order programs.

1 Introduction: Can We Get Two for the Price of One?

In small-step abstract interpretation [4,5,16], there is often a tight correspondence between the concrete and abstract semantics. When one implements a small-step interpreter and then a small-step static analyzer, the correspondence is so obvious that there is a "nagging sense" of duplicated effort—large tracts of code for the analyzer and the interpreter end up looking *almost* identical. Suffering this *déjà vu* long enough leads one to ask:

> Is there a principled method for constructing a sensible abstract interpretation of a small-step concrete semantics automatically?

As we will demonstrate, the answer is *yes*: for any given small-step concrete semantics, there exist "natural" abstract interpretations, and there is a procedure an analysis designer can execute to construct these analyses.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 407–421, 2010.
© Springer-Verlag Berlin Heidelberg 2010

By applying our method to the concrete semantics for continuation-passing style, we end up discovering both known analyses (like k-CFA) and unknown analyses (which take a fundamentally different approach to abstraction of environments and closures). Choice points in the method also end up (quite unexpectedly) providing graph-theoretic explanations for the emergence of properties such as flow-, field- and context-sensitivity (Section 6).

An additional benefit of the method is pedagogical: it adds a more formal dimension to the art of analysis design. One can teach a student *what* abstract interpretation is, and *what* Galois connections are, but this knowledge doesn't make a student an analysis designer any more than rote knowledge of the syntax of Java makes her a programmer. She is still left with the question of *how* to design a static analysis. The method described in this work provides one answer to that question: it constitutes a process students can follow to go from a concrete semantics to an abstract interpreter.

1.1 An Example to Illustrate Correspondence and Redundancy

A brief example informally illustrates the degree to which the abstract semantics resemble the concrete semantics. We point out this resemblance to encourage the idea that the abstract semantics might be synthesized from the concrete semantics. Consider the concrete rule for MOVE in a register machine:

$$([\![\mathit{var} := \mathit{var}']\!] : \boldsymbol{stmt}, \mathit{env}, \mathit{heap}) \Rightarrow (\boldsymbol{stmt}, \mathit{env}[\mathit{var} \mapsto \mathit{env}(\mathit{var}')], \mathit{heap}).$$

The transition moves to the next statement, and updates the environment in the process. Contrast this concrete rule with the "abstract" rule for MOVE:

$$([\![\mathit{var} := \mathit{var}']\!] : \boldsymbol{stmt}, \widehat{\mathit{env}}, \widehat{\mathit{heap}}) \leadsto (\boldsymbol{stmt}, \widehat{\mathit{env}}[\mathit{var} \mapsto \widehat{\mathit{env}}(\mathit{var}')], \widehat{\mathit{heap}}).$$

This rule is suspiciously similar to the concrete one. In fact, the rules are so similar that presenting them both in a technical paper begs charges of redundancy. Implementing them both in code looks like the "copy-and-paste" anti-pattern.

With other rules, the correspondence is less direct. Consider the concrete rule for pointer assignment:

$$([\![*\mathit{var} := \mathit{var}']\!] : \boldsymbol{stmt}, \mathit{env}, \mathit{heap}) \Rightarrow (\boldsymbol{stmt}, \mathit{env}, \mathit{heap}[\mathit{env}(\mathit{var}) \mapsto \mathit{env}(\mathit{var}')]),$$

and its abstract counterpart:

$$\frac{\hat{a} \in \widehat{\mathit{env}}(\mathit{var})}{([\![*\mathit{var} := \mathit{var}']\!] : \boldsymbol{stmt}, \widehat{\mathit{env}}, \widehat{\mathit{heap}}) \leadsto (\boldsymbol{stmt}, \widehat{\mathit{env}}, \widehat{\mathit{heap}} \sqcup [\hat{a} \mapsto \widehat{\mathit{env}}(\mathit{var}')]).}$$

In contrast with the concrete rule, the abstract rule is nondeterministic—there is one subsequent state for each possible abstract address to which the machine may write. The abstract rule also changed from functional extension to join for updating the heap. Staring at the similarities, it *feels* like there should be a principled method that can figure out where to introduce the nondeterminism and where to swap functional extension for join.

1.2 The Two-Step Method: Snipping and Trickling

We will describe a process for converting a small-step concrete semantics into a parameterized abstract semantics. At high level, the process has two steps:

1. The first step **snips** recursive structure out of concrete state-space. While state-spaces with recursive structure *can* be abstracted, it's much easier to abstract state-spaces without recursive structure. To perform the "snip," we view the concrete state-space as a dependence graph. Snipping selectively cuts cycle-forming edges in this graph. Each cut induces a corresponding store-passing-style transformation [17] of the concrete semantics.
2. The second step **trickles** abstraction up the concrete state-space, starting with the leaves of the DAG left over from the snipping operation. The designer must choose a specific abstraction for these leaves. Then, to automate remainder of the process, we recursively apply inference rules that form Galois connections [5]. A Galois connection inference rule has the form, "If the structures X and Y form a Galois connection, the structure $F(X, Y)$ is also a Galois connection," for some functor F. Consequently, these inference rules "trickle up" abstraction from the leaves of the concrete state-space. Once the rules infer a top-level Galois connection between concrete and abstract states, we can calculate the "optimal" abstract interpretation.[1]

The rationale for these two steps comes from an observation on the design of abstract interpretations—finite abstract state-spaces are easier to work with, because no widening is necessary in order to achieve termination. Yet, in order for a small-step semantics to describe a Turing-complete system, the state-space for the small-step semantics must have infinite size. Thus, the motivation for the two-step process is to effect a systematic compaction from an infinite to a finite state-space.

The first step (snipping) exposes the source of the unboundedness of the concrete state-space; it then isolates this unboundedness to the leaf nodes in a dependence graph over the state-space. The second step (trickling) starts by abstracting these leaf nodes into finite sets. Because the snipped concrete state-space lacks recursion, if the abstractions on these leaves are finite, the resulting abstract state-space is also finite.

2 Continuation-Passing-Style λ-Calculus

For the sake of grounding our discussion in specific examples, we'll look at the continuation-passing style λ-calculus (CPS). We will gradually transform the

[1] The word *optimal* has to be qualified: optimal under what constraints? With Galois connections [5], the calculated analysis is *optimal* with respect to the specific abstraction embodied by the Galois connection. Every Galois connection implies many sound analyses, but only one of these is the most precise, and this analysis can be calculated by composing the concretization function with the concrete semantics and again with the abstraction function. That is, the optimal analysis appears to concretize the input, run the exact semantics, and then abstract the output.

concrete semantics for CPS into several abstract interpreters. The grammar for (pure) CPS is conveniently small:

$$\begin{aligned} f, e \in \mathsf{Exp} &::= \mathsf{Var} + \mathsf{Lam} \\ lam \in \mathsf{Lam} &::= (\lambda\ (v_1 \ldots v_n)\ call) \\ v \in \mathsf{Var} &\text{ is a set of identifiers} \\ call \in \mathsf{Call} &::= (f\ e_1 \ldots e_n). \end{aligned}$$

A textbook concrete (small-step) state-space (Σ) for pure CPS is also simple:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times Env \\ \rho \in Env &= \mathsf{Var} \rightharpoonup Clo \\ clo \in Clo &= \mathsf{Lam} \times Env. \end{aligned}$$

And, the small-step transition relation, $(\Rightarrow) \subseteq \Sigma \times \Sigma$ needs but one rule:

$$\begin{aligned} ([\![(f\ e_1 \ldots e_n)]\!], \rho) \Rightarrow (call, \rho''), &\text{ where} \\ (lam, \rho') &= \mathcal{E}(f, \rho) \\ lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call)]\!] \\ \rho'' &= \rho'[v_i \mapsto \mathcal{E}(e_i, \rho)], \end{aligned}$$

where the argument evaluator $\mathcal{E} : \mathsf{Exp} \times Env \rightharpoonup Clo$ evaluates an expression in the context of an environment:

$$\begin{aligned} \mathcal{E}(v, \rho) &= \rho(v) \\ \mathcal{E}(lam, \rho) &= (lam, \rho). \end{aligned}$$

3 A Naïve Attempt: "Throw Hats on Everything"

At first glance, it appears that the only change between concrete and abstract semantics is typographical: hats appear on all of the abstract domains (and the ranges of some functions become, somewhat mysteriously, power domains). Inspired by this observation, we can try it with the domains for continuation-passing style, to arrive at an abstract state-space $\hat{\Sigma}$:

$$\begin{aligned} \hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \\ \hat{\rho} \in \widehat{Env} &= \mathsf{Var} \to \mathcal{P}\left(\widehat{Clo}\right) \\ \widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env}. \end{aligned}$$

But, there is an obvious problem with this "abstract" state-space: it's infinite, because closures contain environments, and environments contain closures. Moreover, a structural abstraction function defined on the concrete state-space isn't

well-founded; there is always the possibility (in theory) that it will encounter an infinite closure, such as clo_∞:

$$clo_\infty = (lam, [v \mapsto \{clo_\infty\}]).$$

Abstract interpreters typically operate over finite state-spaces in order to guarantee termination. For infinite abstract state-spaces, widening can accelerate and guarantee convergence, but a widening operator has to be defined on a case-by-case basis. Constructing an appropriate widening operator is not a process that can be fully mechanized; it requires creativity and intuition. And, in this case, there is no obvious widening operator.

Instead of widening, we choose to eliminate recursion from the state-space through an automatable process called "snipping the knots." Once recursion is eliminated from the concrete state-space, we can systematically transform it into an abstract state-space, starting with its leaves and abstracting upward.

4 Step 1: Snipping the Knots with Store-Passing Style

Recursive structures pose problems with well-foundedness for mathematicians. Because they are difficult to abstract "directly," they also pose a problem for abstract interpretation. Yet, in computer programming, recursive structures—even infinitely recursive structures—are neither uncommon nor troublesome. Every first-year computer science student knows how to build recursive data structures: pointers.

If we view a semantics as an interpreter, then we can exploit this freshman insight to eliminate recursion from mathematical structures as well—we can introduce a store and pointers into a small-step semantics. Specifically, we can use an off-the-shelf store-passing style transformation of the concrete semantics [17], and then thread recursive structure through the store.

To prepare for store-passing style, we represent the concrete definition of the state-space as a graph with edges from uses to definitions of each set (Figure 1). For example, in CPS, we add edges from the node Σ to the node Call and to the node Env, because the definition of the set Σ[2] refers to both Call and Env; for the same reason, we add edges from the node Clo to the node Lam and to the node Env.[3] Once in dependence-graph form, we must choose a set of edges to "snip" in order to eliminate cycles from the graph.

To eliminate cycles in the concrete state-space for CPS (Figure 1), we can snip this graph in either of two places: we can snip the edge from the node Clo

[2] $\Sigma = \mathsf{Call} \times Env$.

[3] The observant reader might wonder why we omit dependence edges between syntax nodes, *e.g.*, from Lam to Call and *vice versa*. In fact, we could add them. However, we will only operate on programs of finite size, and on subterms of the original program. As a result, syntax never contributes to the unboundedness of the concrete state-space; hence, there is no reason to snip these edges. If we used a substitution-/reduction-based concrete semantics, which could introduce new terms during execution, then we would have to add and snip these edges as well.

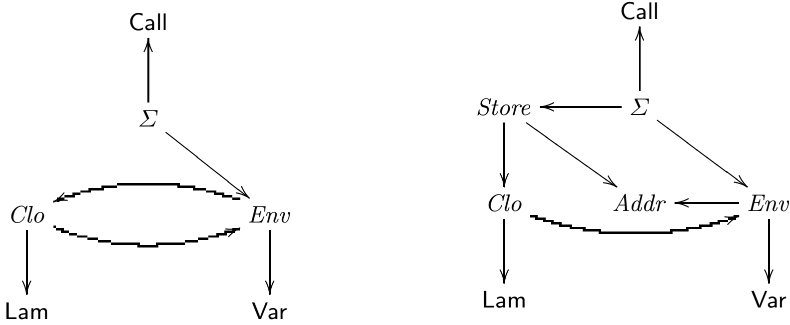


Fig. 1. A dependence graph of the concrete state-space for CPS before a snip (left) and after snipping the $Env \to Clo$ edge (right). After the snip, there are no longer cycles in the dependence graph.

to the node *Env*, or we can snip the edge from the node *Env* to the node *Clo*. It doesn't matter whether we snip one edge or both; the final result will still be a sound abstract interpretation. Snipping the $Env \to Clo$ edge will end up giving us k-CFA [18,19]. Snipping the $Clo \to Env$ edge will end up giving us a novel and interesting hierarchy of control-flow analyses which, to the author's knowledge, has not appeared elsewhere.

4.1 Making a Snip

To make a snip, we need to add a store to the concrete state-space, and then thread this store through the transition relation. To snip an edge going from a set A to a set B, we redirect the snipped edge from its original target to a newly created (infinite) set of addresses *Addr*. We then add the original target to the range of the store *Store*, so that:

$$\sigma \in \mathit{Store} = \mathit{Addr} \rightharpoonup B.$$

Before performing a standard store-passing style transformation on the semantics, the store is made a component of each state.

4.2 Option 1: Snipping *Env* → *Clo*

Snipping the $Env \to Clo$ edge of the CPS semantics and applying the naïve, mechanical store-passing transform to the concrete semantics yields the state-space dependence graph in Figure 1 and the following state-space:

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Clo} \\
a \in \mathit{Addr} & \text{ is an infinite set of addresses,}
\end{aligned}$$

and a new transition rule:

$$\begin{aligned}
([\![(f\ e_1 \ldots e_n)]\!], \rho, \sigma) &\Rightarrow (call, \rho'', \sigma''), \text{ where} \\
((lam, \rho'), \sigma'_0) &= \mathcal{E}((f, \rho), \sigma) \\
lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call)]\!] \\
a_1, \ldots, a_n &\notin dom(\sigma'_0) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
(clo_i, \sigma'_i) &= \mathcal{E}((e_i, \rho), \sigma'_{i-1}) \\
\sigma'' &= \sigma'_n[a_i \mapsto clo_i],
\end{aligned}$$

where the argument evaluator $\mathcal{E} : (\mathsf{Exp} \times Env) \times Store \rightharpoonup (Clo \times Store)$ evaluates an expression in the context of an environment and a store, to return a value and a store:

$$\begin{aligned}
\mathcal{E}((v, \rho), \sigma) &= (\sigma(\rho(v)), \sigma) \\
\mathcal{E}((lam, \rho), \sigma) &= ((lam, \rho), \sigma).
\end{aligned}$$

Cleaning up with useless-variable elimination. Applying useless-variable elimination [20] to the transformed semantics (again treating the semantics like an interpreter) picks up on the fact that the argument evaluator never modifies the store, which leads to a cleaner transition relation:

$$\begin{aligned}
([\![(f\ e_1 \ldots e_n)]\!], \rho, \sigma) &\Rightarrow (call, \rho'', \sigma'), \text{ where} \\
(lam, \rho') &= \mathcal{E}(f, \rho, \sigma) \\
lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call)]\!] \\
a_1, \ldots, a_n &\notin dom(\sigma) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
clo_i &= \mathcal{E}(e_i, \rho, \sigma) \\
\sigma' &= \sigma[a_i \mapsto clo_i],
\end{aligned}$$

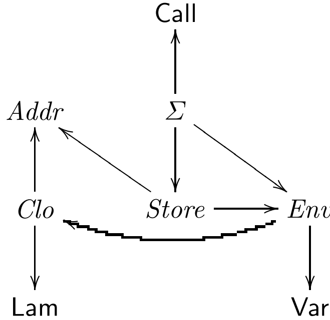
where the argument evaluator $\mathcal{E} : \mathsf{Exp} \times Env \times Store \rightharpoonup Clo$ evaluates an expression in the context of an environment and a store to return a value:

$$\begin{aligned}
\mathcal{E}(v, \rho, \sigma) &= \sigma(\rho(v)) \\
\mathcal{E}(lam, \rho, \sigma) &= (lam, \rho).
\end{aligned}$$

4.3 Option 2: Snipping *Clo* → *Env*

The other option for eliminating recursion is to snip the *Clo* → *Env* edge. This snip leads to a family of analyses with a character unlike any in the published literature on higher-order flow analysis.

Snipping this edge and performing the store-passing transform leads to the following state-space dependence diagram:



and the state-space:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Addr} \\ \sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Env} \\ a \in \mathit{Addr} & \text{ is an infinite set of addresses,} \end{aligned}$$

and the following transition rule:

$$\begin{aligned} (⟦(f\ e_1 \dots e_n)⟧, \rho, \sigma) &\Rightarrow (\mathit{call}, \rho'', \sigma'), \text{ where} \\ a &\notin \mathit{dom}(\sigma) \\ \sigma' &= \sigma[a \mapsto \rho] \\ (\mathit{lam}, a') &= \mathcal{E}(f, a, \sigma') \\ \mathit{lam} &= ⟦(\lambda\ (v_1 \dots v_n)\ \mathit{call})⟧ \\ \mathit{clo}_i &= \mathcal{E}(e_i, a, \sigma') \\ \rho'' &= (\sigma(a'))[v_i \mapsto \mathit{clo}_i], \end{aligned}$$

where the argument evaluator $\mathcal{E} : \mathsf{Exp} \times \mathit{Addr} \times \mathit{Store} \rightharpoonup \mathit{Clo}$ evaluates an expression in the context of an environment's address and a store to return a value:

$$\begin{aligned} \mathcal{E}(v, a, \sigma) &= \sigma(a)(v) \\ \mathcal{E}(\mathit{lam}, a, \sigma) &= (\mathit{lam}, a). \end{aligned}$$

4.4 Optional Snips

Of course, one can also snip non-cycle-forming edges. Under the next stage in the method (trickle-up abstraction), these optional snips manifest themselves as knobs that tune some well-known properties such as field-sensitivity (if one snips

the $Env \to$ Var edge) and flow-sensitivity (if one snips the $\Sigma \to$ Call edge). Yet other snips (such as the $Clo \to$ Lam edge) create knobs for tuning the precision and speed of the analysis which don't appear anywhere in the literature.

Finally, we point out that one can snip as many or as few edges in the dependence graph as desired, so long as the resulting dependence graph is acyclic.

5 Step 2: Trickling Up Abstraction

Once snips have eliminated recursive structure from the concrete state-space (Σ), we need (1) an abstract state-space ($\hat{\Sigma}$), and (2) a Galois connection between the concrete state-space and the abstract state-space ($\mathcal{P}(\Sigma) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\hat{\Sigma})$). Once we have the Galois connection, a foundational result by the Cousots [5] enables us to *calculate* an "optimal" small-step abstract transition relation: $(\leadsto) = \alpha \circ (\Rightarrow) \circ \gamma$.

5.1 Abstracting the Leaves of the State-Space Dependence Graph

To generate the abstract state-space, we focus initially on the leaves of the dependence graph for the concrete state-space. We require that the analysis designer choose a finite set $\hat{A}$ for each leaf node A; these finite sets will become the leaves of the abstract state-space. For each concrete leaf set A, the analysis designer must also specify an extraction function $\eta : A \to \hat{A}$ that maps a concrete element to an abstract element. Once the extraction function is fixed, we can automate the synthesis of the abstract state-space with inference rules that build structural Galois connections.

It is straightforward to convert an extraction function into a Galois connection [13]. Specifically, given a surjective map $\eta : A \to \hat{A}$, the structure $(\mathcal{P}(A), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{A}), \subseteq)$, where:

$$\alpha(S) = \{\eta(a) : a \in S\}$$
$$\gamma(\hat{S}) = \left\{a : \hat{a} \in \hat{S} \text{ and } \eta(a) = \hat{a}\right\},$$

forms a Galois connection.

In practice, snipping and store-passing style transforms will leave an infinite leaf node in the form of the set of addresses. In this case, the extraction function on addresses fixes the polyvariance and the context-sensitivity of the analysis [9].

5.2 Recursively Constructing the Abstract State-Space

To synthesize the abstract state-space automatically, we will utilize inference rules. These inference rules will build up structural Galois connections. In particular, these rules will take the Galois connections defined on leaves, and percolate them up to a top-level Galois connection over sets of states.

Most of the inference rules have the form "if structures $X_1, X_2, \ldots, X_n$ are Galois connections, then $F(X_1, X_2, \ldots, X_n)$ is also a Galois connection (for some functor F)."

Example 1. Given Galois connections $(A, \sqsubseteq_A) \xleftrightarrow[\alpha]{\gamma} (\hat{A}, \sqsubseteq_{\hat{A}})$ and $(B, \sqsubseteq_B) \xleftrightarrow[\alpha']{\gamma'} (\hat{B}, \sqsubseteq_{\hat{B}})$, the product Galois connection is the structure $(A \times B, \sqsubseteq_{A \times B}) \xleftrightarrow[\alpha'']{\gamma''} (\hat{A} \times \hat{B}, \sqsubseteq_{\hat{A} \times \hat{B}})$, where:

$$\alpha''(a, b) = (\alpha(a), \alpha'(b))$$
$$\gamma''(\hat{a}, \hat{b}) = (\gamma(a), \gamma'(b)).$$

For the sake of mechanizing the process, we phrase the definitions of structural Galois connections as inference rules taking us from less-structured Galois connection to a more-structured one; for example:

$$\frac{(A, \sqsubseteq_A) \xleftrightarrow[\alpha]{\gamma} (\hat{A}, \sqsubseteq_{\hat{A}}) \qquad (B, \sqsubseteq_B) \xleftrightarrow[\alpha']{\gamma'} (\hat{B}, \sqsubseteq_{\hat{B}})}{(A \times B, \sqsubseteq_{A \times B}) \xleftrightarrow[\alpha'']{\gamma''} (\hat{A} \times \hat{B}, \sqsubseteq_{\hat{A} \times \hat{B}}).}$$

5.3 Galois Inference Rules

In this work, we use the inference rules sketched in Figure 2 in addition to the "standard" structural Galois connections found in Nielson *et al.* [13]. (For brevity, we omit defining new concretization and abstraction maps in each rule.)

$$(\mathcal{P}(A), \sqsubseteq_1) \xleftrightarrow[\lambda S.S]{\lambda S.S} (\mathcal{P}(A), \sqsubseteq_1) \qquad \text{(power identity)}$$

$$\frac{(\mathcal{P}(A), \sqsubseteq_1) \xleftrightarrow[\alpha]{\gamma} (\mathcal{P}(\hat{A}), \sqsubseteq_2) \qquad (\mathcal{P}(B), \sqsubseteq_1') \xleftrightarrow[\alpha']{\gamma'} (\mathcal{P}(\hat{B}), \sqsubseteq_2')}{(\mathcal{P}(A \times B), \sqsubseteq_1'') \xleftrightarrow[\alpha'']{\gamma''} (\mathcal{P}(\hat{A} \times \hat{B}), \sqsubseteq_2'')} \qquad \text{(power product)}$$

$$\frac{(\mathcal{P}(Y), \sqsubseteq_1) \xleftrightarrow[\alpha]{\gamma} (\mathcal{P}(\hat{Y}), \sqsubseteq_2)}{(\mathcal{P}(X \to Y), \sqsubseteq_1'') \xleftrightarrow[\alpha']{\gamma'} (\mathcal{P}(X \to \hat{Y}), \sqsubseteq_2'')} \qquad \text{(image)}$$

$$\frac{(\mathcal{P}(X), \sqsubseteq_1) \xleftrightarrow[\alpha]{\gamma} (\hat{X}, \sqsubseteq_2)}{(\mathcal{P}(X), \sqsubseteq_1) \xleftrightarrow[\alpha']{\gamma'} (\mathcal{P}(\hat{X}), \sqsubseteq_2')} \qquad \text{(power lift)}$$

$$\frac{(\mathcal{P}(X), \sqsubseteq_1) \xleftrightarrow[\alpha]{\gamma} (\mathcal{P}(\hat{X}), \sqsubseteq_2) \qquad (\mathcal{P}(Y), \sqsubseteq_1') \xleftrightarrow[\alpha']{\gamma'} (\mathcal{P}(\hat{Y}), \sqsubseteq_2')}{(\mathcal{P}(X \times Y), \sqsubseteq_1'') \xleftrightarrow[\alpha'']{\gamma''} (\mathcal{P}(\hat{X} \times \hat{Y}), \sqsubseteq_2'')} \qquad \text{(function)}$$

Fig. 2. Structural inference rules for generating an abstract-state space. Once a designer specifies a Galois connection over the leaves of the concrete state-space, these inference rules construct an abstract state-space and corresponding abstraction/concretization functions.

5.4 Synthesizing an Abstract Interpretation for CPS (Option 1)

Returning to the CPS semantics in which we snipped the $Env \to Clo$ edge and defining an extraction function on addresses $\eta : Addr \to \widehat{Addr}$, we can recursively apply inference rules for Galois connections that lead us to a Galois connection $(\mathcal{P}(\Sigma), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\hat{\Sigma}), \sqsubseteq_{\mathcal{P}(\hat{\Sigma})})$ for the top-level state-space:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \widehat{Clo} \\
\hat{a} \in \widehat{Addr} & \text{ is a finite set of addresses.}
\end{aligned}$$

The function $\alpha : \mathcal{P}(\Sigma) \to \mathcal{P}(\hat{\Sigma})$ encodes the synthesized abstraction map:

$$\begin{aligned}
\alpha \{(call, \rho, \sigma)\} &= \{(call, \alpha(\rho), \alpha(\sigma))\} \\
\alpha(\rho) &= \lambda v.\alpha(\rho(v)) \\
\alpha(\sigma) &= \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a)) \\
\alpha(lam, \rho) &= \{(lam, \alpha(\rho))\} \\
\alpha(a) &= \eta(a).
\end{aligned}$$

Because we have a Galois connection, we can calculate an approximation of the "optimal" abstract transition relation, $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$:

$$\begin{aligned}
\overbrace{([\![(f\ e_1 \dots e_n)]\!], \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \leadsto \overbrace{(call, \hat{\rho}'', \hat{\sigma}')}^{\hat{\varsigma}'}, \text{ where} \\
(lam, \hat{\rho}') \in \hat{\mathcal{E}}(f, \hat{\rho}, \hat{\sigma}) \\
lam &= [\![(\lambda\ (v_1 \dots v_n)\ call)]\!] \\
\hat{a}_i &= \widehat{alloc}(v_i, \hat{\varsigma}) \\
\hat{\rho}'' &= \hat{\rho}'[v_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{E}}(e_i, \hat{\rho}, \hat{\sigma})],
\end{aligned}$$

where the argument evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \times Env \times \widehat{Store} \rightharpoonup \widehat{Clo}$ evaluates an expression in the context of an environment and a store to return a value:

$$\begin{aligned}
\hat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(v)) \\
\hat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) &= \{(lam, \hat{\rho})\}.
\end{aligned}$$

We also introduced the abstract address-allocation function $\widehat{alloc} : \mathsf{Var} \times \hat{\Sigma} \to \widehat{Addr}$. (The concrete semantics selected addresses nondeterministically from outside the domain of the store.) According to Might's *A Posteriori* Soundness Theorem [9], *any* abstract address allocator leads to a sound abstract interpretation.

Example 2. For example, a simple, monovariant address allocator chooses the variable itself for the abstract address:

$$\widehat{Addr} = \mathsf{Var}$$
$$\widehat{alloc}(v, \hat{\varsigma}) = v,$$

which leads to an abstract interpretive formulation of 0CFA.

5.5 Synthesizing an Abstract Interpretation for CPS (Option 2)

Recall that the other option for eliminating recursion is to snip the $Clo \to Env$ edge. This snip leads to a family of analyses with a character unlike any in the published literature on higher-order flow analysis.

Snipping this edge and synthesizing an abstraction leads to the following abstract state-space:

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{Env} \times \widehat{Store}$$
$$\hat{\rho} \in \widehat{Env} = \mathsf{Var} \rightharpoonup \widehat{Clo}$$
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \mathcal{P}\left(\widehat{Env}\right)$$
$$\hat{a} \in \widehat{Addr} \text{ is an finite set of addresses,}$$

and the following transition rule:

$$\overbrace{(\llbracket (f\ e_1 \ldots e_n) \rrbracket, \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \leadsto \overbrace{(call, \hat{\rho}'', \hat{\sigma}')}^{\hat{\varsigma}'}, \text{ where}$$
$$\hat{a} = \widehat{alloc}(\hat{\varsigma})$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\rho}]$$
$$(lam, \hat{a}') \in \hat{\mathcal{E}}(f, \hat{a}, \hat{\sigma}')$$
$$lam = \llbracket (\lambda\ (v_1 \ldots v_n)\ call) \rrbracket$$
$$\widehat{clo}_i = \hat{\mathcal{E}}(e_i, \hat{a}, \hat{\sigma}')$$
$$\hat{\rho}'' = (\hat{\sigma}(\hat{a}'))[v_i \mapsto \widehat{clo}_i],$$

where the argument evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{Addr} \times \widehat{Store} \to \mathcal{P}(\widehat{Clo})$ evaluates an expression in the context of an environment's address and a store to return a value:

$$\hat{\mathcal{E}}(v, \hat{a}, \hat{\sigma}) = \{\hat{\rho}(v) : \hat{\rho} \in \hat{\sigma}(a)\}$$
$$\hat{\mathcal{E}}(lam, \hat{a}, \hat{\sigma}) = \{(lam, a)\}\,.$$

6 Flow-Sensitivity, Field-Sensitivity and Context-Sensitivity

We mentioned earlier that snipping different edges could lead to different knobs for tuning the precision of the analysis. Properties such as flow-, field- and context-sensitivity emerge as the result of extra snips in the original dependence graph, and their degree can be tuned by the extraction function required to form the Galois connection.

Flow-sensitivity. Consider, for example, snipping the $\Sigma \rightarrow \mathsf{Call}$ edge in the CPS semantics. That is, instead of a state having the structure $\varsigma = (call, \ldots)$, it will have the structure $\varsigma' = (a_{call}, \ldots, \sigma)$, where $call = \sigma(a_{call})$. Thus, call sites become addressable values, and to abstract, one must define an extraction function. This extraction function on addresses of call sites creates a concrete leaf node that, under the second step, maps to "abstract call sites." If all concrete call sites abstract to the same abstract call site, *i.e.* $\eta(a_{call}) = \hat{a}_0$ for all call site addresses a_{call}, then the optimal analysis becomes completely flow-insensitive. If, on the other hand, the extraction function is the identity function, then the optimal analysis is completely flow-sensitive. The nature of the abstraction from concrete to abstract call sites precisely captures the flow-sensitivity of the resulting analysis.

Field-sensitivity. In higher-order languages, environments play the role of structures. Thus, for CPS, field-sensitivity manifests as the degree to which variables in a given environment have the same abstract address. To create a Galois connection that tunes this parameter, we need only snip the $Env \rightarrow \mathsf{Var}$ edge in the concrete dependence graph. Once again, a singleton extraction map leads to field-insensitivity, and an identity extraction map leads to field-sensitivity.

Context-sensitivity and polyvariance. The term *polyvariance* refers to the number of abstractions (variants) for a given variable (or allocation site). Monovariant analyses like 0CFA have only one abstract address for each variable. Typically, context-sensitivity determines polyvariance by carving up the abstract variants of a variable according the contexts in which it is bound. Thus, to tune polyvariance, snip the $Env \rightarrow Clo$ edge in the concrete state-space graph, and adjust the extraction function for the resulting Galois connection.

7 Related Work

This work draws most directly on three lines of research: abstract interpretation [4], formal semantics [17] and Galois connections [5]. The programmatic transformation of formal semantics dates to work by Reynolds [15]. More recent work by Danvy *et al.* has shown that formal semantics are highly amenable to program transformations and that it is possible to automatically convert denotational semantics into operational semantics and *vice versa* [2,3,1,6,7,8]. These techniques, combined with ours, should permit the mechanizable construction of static analyzers for a wider variety of formal semantics paradigms.

The Cousots' foundational work on Galois connections marks the earliest attempts to mechanize the process of constructing an abstract interpretation [5]. Given a Galois connection $X \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X}$, it is possible to calculate the optimal abstract image of a concrete function $f : X \to X$ as $\hat{f} = \alpha \circ f \circ \gamma$. Our work advances the Cousots' original work by automating the construction of the Galois connection itself using inference rules. There have been additional attempts to automate parts of the process of constructing an abstraction; most recently, work by Qian *et al.* has focused on constructing minimal abstractions that lead to completeness [14].

Our running example on the abstraction of continuation-passing style lambda calculus is an instance of the long line of work on higher-order control-flow analysis [19]. The first family of analyses we derived corresponds to universal framework for k-CFA-like analyses [12]. The second family of analyses we derived is difficult to place in relation to existing analyses. To begin, it is the only analysis which does not abstract the range of environments. This gives it the unique feature that variable look-up in such an analysis yields exactly one abstract closure. It also opens up the prospect of using techniques such as abstract counting directly on environment addresses in order to perform must-alias analysis [10,11].

8 Summary and Conclusion

We have presented a two-step method for converting a small-step concrete semantics into an abstract interpretation. The first step eliminates recursive structure from the concrete state-space by snipping edges in the dependence graph of the concrete state-space; the second step trickles abstraction up the leaves of the newly re-factored concrete state-space. Inference rules over structural Galois connections synthesize the abstract state-space, and a Galois connection between concrete and abstract states at the same time. The synthesized Galois connection also determines the optimal abstract interpretation. By snipping additional edges in the concrete dependence graph, these snips turn into knobs for tuning flow-, field- and context-sensitivity under abstraction. The immediate payoff of this method in our work was (1) a re-affirmation that k-CFA is, in some sense a fundamental technique, and (2) a new family of analyses based on a novel abstraction of environments.

References

1. A functional correspondence between evaluators and abstract machines. ACM Press, New York (2003)
2. Ager, M., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theoretical Computer Science 342(1),149–172 (2005)
3. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Processing Letters 90(5), 223–232 (2004)

4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages pp. 238–252. ACM Press, New York (1977)
5. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 269–282. ACM Press, New York (1979)
6. Danvy, O., Millikin, K.: A rational deconstruction of landin's secd machine with the j operator. Logical Methods in Computer Science 4(4) (November 2008)
7. Danvy, O., Millikin, K.: Refunctionalization at work. Science of Computer Programming 74(8), 534–549 (2009)
8. Midtgaard, J.: Transformation, Analysis, and Interpretation of Higher-Order Procedural Programs. PhD thesis, University of Aarhus (2007)
9. Might, M., Manolios, P.: A posteriori soundness for non-deterministic abstract interpretations. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 260–274. Springer, Heidelberg (2009)
10. Might, M., Shivers, O.: Improving flow analyses via γcfa: Abstract garbage collection and counting. In: ICFP 2006: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, pp. 13–25. ACM, New York (2006)
11. Might, M., Shivers, O.: Exploiting reachability and cardinality in higher-order flow analysis. Journal of Functional Programming, Special Double Issue 18(5-6), 821–864 (2008)
12. Nielson, F., Nielson, H.R.: Infinitary control flow analysis: a collecting semantics for closure analysis. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 332–345. ACM, New York (1997)
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Corrected ed. Springer, Heidelberg (October 1999)
14. Qian, J., Zhao, L., Cai, G., Gu, T.: Automatic construction of complete abstraction by abstract interpretation. In: ICIS 2009: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science, Washington, DC, USA, pp. 927–932. IEEE Computer Society, Los Alamitos (2009)
15. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: ACM 1972: Proceedings of the ACM Annual Conference, pp. 717–740. ACM, New York (1972)
16. Schmidt, D.A.: Abstract interpretation of small-step semantics. In: Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, London, UK, pp. 76–99. Springer, Heidelberg (1997)
17. Scott, D., Strachey, C.: Towards a formal semantics, pp. 197–220 (1966)
18. Shivers, O.: Control flow analysis in Scheme. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, vol. 23, pp. 164–174. ACM, New York (July 1988)
19. Shivers, O. G.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University (1991)
20. Wand, M., Siveroni, I.: Constraint systems for useless variable elimination. In: POPL 1999: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 291–302. ACM, New York (1999)

Points-to Analysis as a System of Linear Equations

Rupesh Nasre and Ramaswamy Govindarajan

Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India
{nasre,govind}@csa.iisc.ernet.in

Abstract. We propose a novel formulation of the points-to analysis as a system of linear equations. With this, the efficiency of the points-to analysis can be significantly improved by leveraging the advances in solution procedures for solving the systems of linear equations. However, such a formulation is non-trivial and becomes challenging due to various facts, namely, multiple pointer indirections, address-of operators and multiple assignments to the same variable. Further, the problem is exacerbated by the need to keep the transformed equations linear. Despite this, we successfully model all the pointer operations. We propose a novel inclusion-based context-sensitive points-to analysis algorithm based on prime factorization, which can model all the pointer operations. Experimental evaluation on SPEC 2000 benchmarks and two large open source programs reveals that our approach is competitive to the state-of-the-art algorithms. With an average memory requirement of mere 21MB, our context-sensitive points-to analysis algorithm analyzes each benchmark in 55 seconds on an average.

1 Introduction

Points-to analysis enables several compiler optimizations and remains an important static analysis technique. Enormous growth of code bases in proprietary and open source software systems demands scalability of static analyses over billions of lines of code. Several points-to analysis algorithms have been proposed in literature that make this research area rich in content [1,26,5,2,18].

A points-to analysis is a method of statically determining whether two pointers may point to the same location at runtime. The two pointers are then said to be aliases of each other. For analyzing a general purpose C program, it is sufficient to consider all pointer statements of the following forms: address-of assignment ($\mathtt{p} = \&\mathtt{q}$), copy assignment ($\mathtt{p} = \mathtt{q}$), load assignment ($\mathtt{p} = *\mathtt{q}$) and store assignment ($*\mathtt{p} = \mathtt{q}$) [25].

A flow-insensitive analysis ignores the control flow in the program and, in turn, assumes that the statements could be executed in any order. A context-sensitive analysis takes into consideration the calling context of a statement while computing the points-to information. Storing complete context information can exponentially blow up the memory requirement and increase analysis time, making the analysis non-scalable for large programs.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 422–438, 2010.
© Springer-Verlag Berlin Heidelberg 2010

It has been established that flow-sensitivity does not add a significant precision over a flow-insensitive analysis [16]. Therefore, we consider context-sensitive flow-insensitive points-to analysis in this paper.

A flow-insensitive points-to analysis iterates over a set of constraints obtained from points-to statements until it reaches a fixpoint. We observe that this phenomenon is similar in spirit to obtaining a solution to a system of linear equations. Each equation defines a constraint on the feasible solution and a linear solver progressively approaches the final solution in an iterative manner. Similarly, every points-to statement forms a constraint on the feasible points-to information and every iteration of a points-to analysis *refines* the points-to information obtained over the previous iteration. We exploit this similarity to *map* the input source program to a set of linear constraints, solve it using a standard linear equation solver and *unmap* the results to obtain the points-to information. As we show in the next section, a naive approach of converting points-to statements into a linear form faces several challenges due to (i) the distinction between ℓ-value and r-value in points-to statements, (ii) multiple dereferences of a pointer and (iii) the same variable defined in multiple statements. We address these challenges with novel mechanisms based on prime factorization of integers.

Major contributions of this paper are as below.

- A novel representation using prime factorization to store points-to facts.
- We transform points-to constraints into a system of linear equations without affecting precision.
- We show the effectiveness of our approach by comparing it with state-of-the-art context-sensitive algorithms using SPEC 2000 benchmarks and two large open source programs (*httpd* and *sendmail*). On an average, our method computes points-to information in 55 seconds using 21 MB memory proving competitive to other methods.

2 Points-to Analysis

In the following subsection, we describe a simple method to convert a set of points-to statements into a set of linear equations. Using it as a baseline, we discuss several challenges that such a method poses. Consequently, in Section 2.2, we introduce our novel transformation that addresses all the discussed issues and present our points-to analysis algorithm (Section 2.3). Later, we extend it for context-sensitivity using an invocation graph based approach (Section 2.4) and prove its soundness in the next section.

2.1 A First-Cut Approach

Consider the set of C statements given in Figure 1(a). Let us define a transformation that translates &x into $\mathtt{x} - \mathtt{1}$ and $*^{\mathtt{k}}\mathtt{x}$ into $\mathtt{x} + \mathtt{k}$, where k is the number of '*'s used in dereferencing. Thus, $*\mathtt{x}$ is transformed as $\mathtt{x} + \mathtt{1}$, $**\mathtt{x}$ as $\mathtt{x} + \mathtt{2}$ and so on. A singleton variable without any operations is copied as it is, thus,

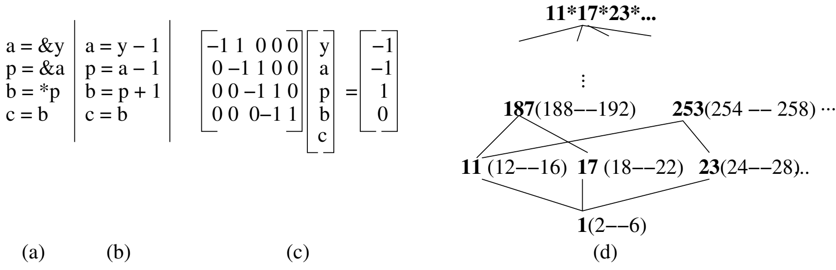


Fig. 1. (a, b, c) Example to illustrate points-to analysis as a system of linear equations. (d) Lattice over the compositions of primes guaranteeing five levels of dereferencing.

x translates to x. The transformed program now looks like Figure 1(b). This becomes a simple linear system of equations that can be written in matrix form $\mathtt{AX} = \mathtt{B}$ as shown in Figure 1(c).

This small example illustrates several interesting aspects. First, the choices of transformation functions for '∗' and '&' are not independent, because '∗' and '&' are complementary operations by language semantics, which should be carried to the linear transformation. Second, selecting $\mathtt{k} = \mathtt{0}$ is not a good choice because we lose information regarding the address-of and dereference operations, resulting in loss of precision in the analysis. Third, every row in matrix A has at most two 1s, i.e., every equation has at most two unknowns. All the entries in matrices A and B are 0, 1 or -1.

Solving the above linear system using a solver yields the following parameterized result: $\mathtt{y} = \mathtt{r}, \mathtt{a} = \mathtt{r} - 1, \mathtt{p} = \mathtt{r} - 2, \mathtt{b} = \mathtt{r} - 1, \mathtt{c} = \mathtt{r} - 1$.

From the values of the variables, we can quickly conclude that a, b and c are aliases. Further, since the value of p is one smaller than that of a, we say that p points to a, and in the same manner, a points to y. Thus, our analysis computes all the points-to information obtained using Andersen's analysis, and is thus *sound*: $\mathtt{y} \rightarrow \{\}, \mathtt{a} \rightarrow \{\mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{y}\}$.

Next, we discuss certain issues with this approach.

Imprecise analysis. Note that our solver also added a few spurious points-to pairs: $\mathtt{p} \rightarrow \{\mathtt{b}, \mathtt{c}\}$. Therefore, the first-cut approach described above gives an *imprecise* result.

Cyclic dependences. Note that each constraint in the above system of equations is of the form $\mathtt{a_i} - \mathtt{a_j} = \mathtt{b_{ij}}$ where $\mathtt{b_{ij}} \in \{0, 1, -1\}$. We can build a constraint graph $\mathtt{G} = (\mathtt{V}, \mathtt{E}, \mathtt{w})$ where

$$\mathtt{V} = \{\mathtt{a_1}, ..., \mathtt{a_n}\} \cup \{\mathtt{a_0}\},\ \mathtt{w}(\mathtt{a_i}, \mathtt{a_j}) = \mathtt{b_{ij}}a,\ \mathtt{w}(\mathtt{a_0}, \mathtt{a_i}) = \mathtt{0} \text{ and}$$
$$\mathtt{E} = \{(\mathtt{a_i}, \mathtt{a_j}) : \mathtt{a_i} - \mathtt{a_j} = \mathtt{b_{ij}} \text{ is a constraint}\} \cup \{(\mathtt{a_0}, \mathtt{a_1}), ..., (\mathtt{a_0}, \mathtt{a_n})\}.$$

The above linear system has a feasible solution *iff* the corresponding constraint graph has no cycle with negative weight [3]. A linear solver would not output any solution for a system with a cycle. Our algorithm uses appropriate variable renaming that allows a standard linear solver to solve such equations.

Inconsistent equations. The above approach fails for multiple assignments to the same variable. For instance, $\texttt{a} = \&\texttt{x}, \texttt{a} = \&\texttt{y}$ is a valid program fragment. However, $\texttt{a} = \texttt{x} - 1, \texttt{a} = \texttt{y} - 1$ does not form a consistent equation system unless $\texttt{x} = \texttt{y}$. This issue is discussed in the context of *bug-finding* [12].

Nonlinear system of equations. One way to handle inconsistent equations is to multiply the constraints having the same unknown to generate a non-linear set of equations. Thus, $\texttt{a} = \&\texttt{x}$ and $\texttt{a} = \&\texttt{y}$ would generate a non-linear constraint $(\texttt{a} - \texttt{x} + 1)(\texttt{a} - \texttt{y} + 1) = 0$. However, non-linear analysis is often more expensive than a linear analysis. Further, maintaining integral solutions across iterations using standard techniques is a difficult task.

Equations versus inequations. The inclusion-based analysis semantics for a points-to statement $\texttt{a} = \texttt{b}$ imply points-to-set(a) $\supseteq$ points-to-set(b). Transforming the statement into an equality $\texttt{a} - \texttt{b} = 0$ instead of an inequality can be imprecise, as equality in mathematics is bidirectional. It is easy to verify, however, that if a set of constraints contains each *ℓ-value* at most once *and* this holds across iterations of the analysis, the solution sets obtained using inequalities and equalities would be the same. We exploit this observation in our algorithm.

Dereferencing. As per our first-cut approach, transformations of points-to statements $\texttt{a} = \&\texttt{b}$ and $*\texttt{a} = \texttt{b}$ would be $\texttt{a} = \texttt{b} - 1$ and $\texttt{a} + 1 = \texttt{b}$ respectively. According to the algebraic semantics, the above equations are equivalent, although the two points-to statements have different semantics. This necessitates one to take care of the *store* constraints separately.

2.2 The Modified Approach

We solve the issues with the above approach with a modified mechanism. Our approach is iterative, and in each iteration, it goes through four major steps, viz., preprocessing, solving the linear system of equations, post-processing and evaluating *special* constraints. We illustrate it using the following example.

$$\texttt{a} = \&\texttt{x}; \texttt{b} = \&\texttt{y}; \texttt{p} = \&\texttt{a}; \texttt{c} = *\texttt{p}; *\texttt{p} = \texttt{b}; \texttt{q} = \texttt{p}; \texttt{p} = *\texttt{p}; \texttt{a} = \texttt{b}.$$

Pre-processing. First, we move *store* constraints from the set of equations to a set of generative constraints (as they *generate* more linear equations) that are processed specially. We proceed with the remaining non-*store* constraints.

Second, all constraints of the form $\texttt{v} = \texttt{e}$ are converted to $\texttt{v} = \texttt{v}_{\texttt{i}-1} \oplus \texttt{e}$. Here, $\texttt{v}_{\texttt{i}-1}$ is the value of the variable $\texttt{v}$ obtained in the last iteration. Initially, $\texttt{v} = \texttt{v}_0 \oplus \texttt{e}$. This transformation ensures monotonicity required for a flow-insensitive points-to analysis. The operator $\oplus$ would be concretized shortly. $\texttt{v}_0$ is a constant, since it is already computed from the previous iteration.

Next, we assign unique prime numbers from a select set $\mathcal{P}$ to the right-hand side expression in each address-of constraint. We defer the definition of $\mathcal{P}$ to a later part of this subsection. Let $\&\texttt{x}$, $\&\texttt{y}$ and $\&\texttt{a}$ be assigned arbitrary prime numbers, say $\&\texttt{x} = 17$; $\&\texttt{y} = 29$; and $\&\texttt{a} = 101$. The addresses of the remaining variables $(\texttt{b}, \texttt{p}, \texttt{q}, \texttt{c})$ are assigned a special sentinel value χ. Further, all the variables

of the form $\mathtt{v}$ and $\mathtt{v_i}$ are assigned an initial r-value of χ. Thus, $\mathtt{x, y, a, b, c, p, q}$ and $\mathtt{x_0}$, $\mathtt{y_0}$, $\mathtt{a_0}$, $\mathtt{b_0}$, $\mathtt{c_0}$, $\mathtt{p_0}$, $\mathtt{q_0}$ equal χ. We keep two-way maps of variables to their r-values and addresses. This step is performed only once in the analysis. In the rest of the paper, the term "address of a variable" refers to the prime number assigned to it by our static analysis.

Next, the dereference $*\mathtt{q}$ in every load statement $\mathtt{p} = *\mathtt{q}$ is replaced by expression $\mathtt{q_{i-1}} + \mathtt{1}$ where $\mathtt{i}$ is the current iteration. Therefore, $\mathtt{c} = *\mathtt{p}$ becomes $\mathtt{c} = \mathtt{c_0} \oplus (\mathtt{p_0}+1)$ and $\mathtt{p} = *\mathtt{p}$ becomes $\mathtt{p} = \mathtt{p_0} \oplus (\mathtt{p_0} + \mathtt{1})$. Note that by generating different versions of the same variable in this manner, we remove cyclic dependences altogether, since variables $\mathtt{v_i}$'s are not dependent on any other variable as they are never *defined* explicitly in the constraints. The renaming is only symbolic and appears only for exposition purposes. Since values from only the previous iteration are required, we simply make a copy $\mathtt{v_{copy}}$ for each variable $\mathtt{v}$ at the start of each iteration.

Last, we rename multiple occurrences of the same variable as an ℓ*-value* in different constraints to convert it to an SSA-like form. For each such renamed variable $\mathtt{v}'$, we store a constraint of the form $\mathtt{v} = \mathtt{v}'$ in a separate merging constraint set. Thus, assignments to $\mathtt{a}$ in $\mathtt{a} = \mathtt{x_0}$ and $\mathtt{a} = \mathtt{b_0}$ are replaced as $\mathtt{a} = \mathtt{x_0}$ and $\mathtt{a}' = \mathtt{b_0}$ and the constraint $\mathtt{a} = \mathtt{a}'$ is added to the merging constraints set. The constraints now look as follows.

Linear constraints: $\mathtt{a} = \mathtt{a_0} \oplus \&\mathtt{x}; \mathtt{b} = \mathtt{b_0} \oplus \&\mathtt{y}; \mathtt{p} = \mathtt{p_0} \oplus \&\mathtt{a};$
$\mathtt{c} = \mathtt{c_0} \oplus (\mathtt{p_0} + \mathtt{1}); \mathtt{q} = \mathtt{q_0} \oplus \mathtt{p}; \mathtt{p}' = \mathtt{p_0} \oplus (\mathtt{p_0} + \mathtt{1}); \mathtt{a}' = \mathtt{a_0} \oplus \mathtt{b}.$
Generative constraints: $*\mathtt{p} = \mathtt{b}.$
Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'.$

Substituting the r-values and the primes for the addresses of variables, we get
$\mathtt{a} = \chi \oplus \mathtt{17}, \mathtt{b} = \chi \oplus \mathtt{29}, \mathtt{p} = \chi \oplus \mathtt{101}, \mathtt{c} = \chi \oplus (\chi + \mathtt{1}),$
$\mathtt{q} = \chi \oplus \mathtt{p}, \mathtt{p}' = \chi \oplus (\chi + \mathtt{1}), \mathtt{a}' = \chi \oplus \mathtt{b}.$

χ and $\oplus$. We unfold the mystery behind the values of χ and $\oplus$ now. The rationale behind replacing the address of every address-taken variable with a prime number is to have a *non-decomposable* element defining the variable. We make use of *prime factorization* of integers to map a value to the corresponding points-to set. The first trivial but important observation towards this goal is that any pointee of any variable has to appear as address taken in at least one of the constraints. Therefore, the only pointees any pointer can have would exactly be the address-taken variables. Thus, a *composition* $\mathtt{v} = \mathtt{v_i} \oplus \mathtt{v_j} \oplus ...$ of the primes $\mathtt{v_i}, \mathtt{v_j}, ...$ representing address-taken variables defines the pointer $\mathtt{v}$ pointing to all these address-taken variables. The composition is defined by operator $\oplus$ and it defines a lattice over the finite set of all the pointers and pointees (Figure 1(d)). The top element $\top$ defines a composition of all address-taken variables ($\mathtt{v_0} \oplus \mathtt{v_1} \oplus ... \oplus \mathtt{v_n}$) and the bottom element $\perp$ defines the empty set. Since we use prime factorization, $\oplus$ becomes the multiplication operator $\times$ and χ is the identity element, i.e., $\mathtt{1}$. The reason behind using $\oplus$ and χ as

placeholders is that it is possible to use an alternative lattice with different $\oplus$ and χ and achieve an equivalent transformation (as long as the equations remain linear). Since every positive integer has a unique prime factorization, we guarantee that the value of a pointer uniquely identifies its pointees. For instance, if $\mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}$ and $\mathtt{b} \rightarrow \{\mathtt{y}, \mathtt{z}, \mathtt{w}\}$, then we can assign primes to $\&\mathtt{x}, \&\mathtt{y}, \&\mathtt{z}, \&\mathtt{w}$ arbitrarily as $\&\mathtt{x} = \mathtt{11}, \&\mathtt{y} = \mathtt{19}, \&\mathtt{z} = \mathtt{5}, \&\mathtt{w} = \mathtt{3}$ and the values of a and b would be calculated as $\mathtt{a} = \mathtt{11} \times \mathtt{19} = \mathtt{209}$ and $\mathtt{b} = \mathtt{19} \times \mathtt{5} \times \mathtt{3} = \mathtt{285}$. Since, 209 can only be factored as $\mathtt{11} \times \mathtt{19}$ and 285 does not have any other factorization than $\mathtt{19} \times \mathtt{5} \times \mathtt{3}$, we can obtain the points-to sets for pointers a and b from the factors.

Unfortunately, prime factorization is not known to be polynomial [19]. Therefore, for efficiency reasons, our implementation keeps track of the factors explicitly. We use a prime-factor-table for this purpose. The prime-factor-table stores all the prime factors of a value. We initially store all the primes p corresponding to the address-taken variables as $\mathtt{p} = \mathtt{p} \times \mathtt{1}$.

Thus, the system of equations now becomes

Linear constraints: $\mathtt{a} = \mathtt{17}; \mathtt{b} = \mathtt{29}; \mathtt{p} = \mathtt{101}; \mathtt{c} = \mathtt{2}; \mathtt{q} = \mathtt{p}; \mathtt{p}' = \mathtt{2}; \mathtt{a}' = \mathtt{b}$.

Generative constraints: $*\mathtt{p} = \mathtt{b}$. Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

Solving the system. Solving the above system of equations using a standard linear solver gives us the following solution.

$\mathtt{a} = \mathtt{17}, \mathtt{b} = \mathtt{29}, \mathtt{p} = \mathtt{101}, \mathtt{c} = \mathtt{2}, \mathtt{q} = \mathtt{101}, \mathtt{p}' = \mathtt{2}, \mathtt{a}' = \mathtt{29}$.

Post-processing. Interpreting the values in the above solution obtained using a linear solver is straightforward except for those of c and $\mathtt{p}'$ (2 is not chosen to be one of the primes.). In the simple case, a value $\mathtt{v} + \mathtt{k}$ denotes $\mathtt{k}^{\mathtt{th}}$ dereference of v. To find v, our method checks each value ϑ in $(\mathtt{v} + \mathtt{k}), (\mathtt{v} + \mathtt{k} - \mathtt{1}), (\mathtt{v} + \mathtt{k} - \mathtt{2}), ...$ in the prime factor table. For the first ϑ that appears in the prime factor table, $\mathtt{v} = \vartheta$ and $\mathtt{k}' = \mathtt{v} + \mathtt{k} - \vartheta$ represents the level of dereferencing. We obtain the prime factors of ϑ from the table, which would correspond to the addresses of variables, reverse-map the addresses to their corresponding variables, then obtain the r-values of the variables from the map whose prime factors would denote the points-to set we want for expression $\mathtt{v} + \mathtt{k}$. Another level of reverse mapping-mapping would be required for $\mathtt{k} = \mathtt{2}$ and so on. We explain dereferencing method (Algorithm 3) later. Note that since our method can handle only a limited number of dereferences (k), the number of iterations required in the dereferencing step is also limited (and is typically small). Therefore, in the example, the value 2 of the variables c and $\mathtt{p}'$ is represented as $\mathtt{1} + \mathtt{1}$ where the second 1 denotes a dereference and the first 1 is the value of the variable being dereferenced. In this case, since $\mathtt{v} = \mathtt{1}$, which is the sentinel χ, its dereference results in an empty set and thus, both c and $\mathtt{p}'$ are assigned a value of 1.

Selection of primes. In general, a value ϑ may be interpreted as $\mathtt{v}_1 + \mathtt{k}_1$ as well as $\mathtt{v}_2 + \mathtt{k}_2$, if the values $\mathtt{v}_1$ and $\mathtt{v}_2$ happen to be close to each other. To avoid this ambiguity, the ranges $(\mathtt{v}_1 ... \mathtt{v}_1 + \mathtt{k})$ and $(\mathtt{v}_2 ... \mathtt{v}_2 + \mathtt{k})$ must be non-overlapping for a fixed k. This is accomplished by careful selection of the prime numbers representing

the address-taken variables. Our analysis selects primes offline and guarantees that a certain $\mathtt{k}$ number of dereferences will never overlap with one another. In fact, we define our method for upto $\mathtt{k}$ levels of dereferencing. Our prime number set $\mathcal{P}$ is also defined for a specific $\mathtt{k}$. More specifically, for any prime numbers $\mathtt{p} \in \mathcal{P}$, the products of any one[1] or more $\mathtt{p}$ are distance more than $\mathtt{k}$ apart. Thus, $|\mathtt{p_i} - \mathtt{p_j}| > \mathtt{k}$ and $|\mathtt{p_i} * \mathtt{p_j} - \mathtt{p_l} * \mathtt{p_m}| > \mathtt{k}$ and $|\mathtt{p_i} * \mathtt{p_j} * \mathtt{p_l} - \mathtt{p_m} * \mathtt{p_n} * \mathtt{p_o}| > \mathtt{k}$ and so on. Note that $\mathcal{P}$ needs to be computed only once, offline. Also, typically, the number of dereferences in real-world programs is very small (< 5). The lattice for the prime number set $\mathcal{P}$ chosen for $k = 5$ is shown in Figure 1(d). Here, the bracketed values, e.g., (12,13,...,16) denote possible dereferencings of a variable which is assigned the value 11.

The next step is to merge the points-to sets of renamed variables, i.e., evaluating merging constraints. This changes $\mathtt{a}$ and $\mathtt{p}$ as $\mathtt{a} = 17 \times 29$ and $\mathtt{p} = 101 \times 1 = 101$.

After merging, we discard all the renamed variables.

Thus, at the end of the first iteration, the points-to set contained in the values is: $x \rightarrow \{\}, y \rightarrow \{\}, a \rightarrow \{x, y\}, b \rightarrow \{y\}, c \rightarrow \{\}, p \rightarrow \{a\}, q \rightarrow \{a\}$.

Evaluating special constraints: The final step is to evaluate the generative constraints and generate more linear constraints. In the first iteration, the store constraint $*\mathtt{p} = \mathtt{b}$ generates the copy constraint $\mathtt{a} = \mathtt{b}$ which already exists in the system. Thus, no new linear constraints are generated. Note that the generative constraints set is retained as more constraints may need to be added in further iterations. At the end of each iteration, our algorithm checks if any variable value is changed since the last iteration. If yes, then another iteration is required.

Subsequent iterations. The constraints, ready for iteration number two, are

Linear constraints: $\mathtt{a} = \mathtt{a_1} \times \&\mathtt{x}; \mathtt{b} = \mathtt{b_1} \times \&\mathtt{y}; \mathtt{p} = \mathtt{p_1} \times \&\mathtt{a};$
$\mathtt{c} = \mathtt{c_1} \times (\mathtt{p_1} + 1); \mathtt{q} = \mathtt{q_1} \times \mathtt{p}; \mathtt{p}' = \mathtt{p_1} \times (\mathtt{p_1} + 1); \mathtt{a}' = \mathtt{a_1} \times \mathtt{b}.$

Generative constraints: $*\mathtt{p} = \mathtt{b}$.

Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

Here, $\mathtt{v_1}$ is the value of the variable $\mathtt{v}$ obtained in iteration 1. Thus the constraints to be solved by the linear solver are:

$\mathtt{a} = 17 \times 29 \times 17, \mathtt{b} = 29 \times 29, \mathtt{p} = 101 \times 101, \mathtt{c} = 101 + 1, \mathtt{q} = 101 \times \mathtt{p},$
$\mathtt{p}' = 101 \times (101 + 1), \mathtt{a}' = 17 \times 29 \times \mathtt{b}.$

The linear solver offers the following solution.

$\mathtt{a} = 17 \times 29 \times 17, \mathtt{b} = 29 \times 29, \mathtt{p} = 101 \times 101, \mathtt{c} = 102, \mathtt{q} = 101 \times 101 \times 101,$
$\mathtt{p}' = 101 \times 102, \mathtt{a}' = 17 \times 29 \times 29 \times 29.$

The solver returns each value as an integer (e.g., 8381) and not as factors (e.g., $17 \times 29 \times 17$). Our analysis finds the prime factors using the prime-factor-table.

Post-processing over the values starts with *pruning the powers* of the values containing repeated prime factors as they do not add any additional points-to information to the solution. Thus,

$\mathtt{a} = 17 \times 29, \mathtt{b} = 29, \mathtt{p} = 101, \mathtt{c} = 102, \mathtt{q} = 101, \mathtt{p}' = 101 \times 102, \mathtt{a}' = 17 \times 29.$

[1] Product of one number is the number itself.

The next step is to dereference variables to obtain their points-to sets. Since, 17, 29, and 101 are directly available in prime factor table, the values of $\mathtt{a}, \mathtt{b}, \mathtt{p}, \mathtt{q}, \mathtt{a}'$ do not require a dereference. In case of c, 102 is not present in prime factor table, so the next value 101 is searched for, which indeed is present in the table. Thus, $(\mathtt{102} - \mathtt{101})$ dereferences are done on 101. Further, 101 reverse-maps to &a and a forward-maps to the r-value $\mathtt{17} \times \mathtt{29}$. Hence $\mathtt{c} = \mathtt{17} \times \mathtt{29}$, suggesting that c points to x and y.

The value of $\mathtt{p}'$ is an interesting case. The solution returned by the solver (10302) is neither a prime number, nor a short offset from the product of primes. Rather, it is a product of a prime and a short offset of the prime. We know that it is the value of variable $\mathtt{p}'$ whose original value was $\mathtt{p}_1 = \mathtt{101}$. This original value is used to find out the points-to set contained in value 10302. To achieve this, our method (always) divides the value obtained by the solver by the original value of the variable. Thus, we get $\mathtt{10302}/\mathtt{101} = \mathtt{102}$. Our method then applies the dereferencing algorithm on 102 to get its points-to set, which, as explained above for c, computes the value $\mathtt{17} \times \mathtt{29}$ corresponding to the points-to set $\{\mathtt{x}, \mathtt{y}\}$. This updates $\mathtt{p}'$ to $\mathtt{101} \times \mathtt{17} \times \mathtt{29}$.

It should be emphasized that our method never checks a number for primality. After prime-factor-table is initially populated with statically defined primes as a multiple of self and unity, a lookup in the table suffices for primality testing.

The next step is to evaluate the merging set to obtain the following.

$\mathtt{a} = \mathtt{17} \times \mathtt{29} \times \mathtt{17} \times \mathtt{29}, \mathtt{p} = \mathtt{101} \times (\mathtt{101} \times \mathtt{17} \times \mathtt{29})$,

which on pruning gives $\mathtt{a} = \mathtt{17} \times \mathtt{29}, \mathtt{p} = \mathtt{17} \times \mathtt{29} \times \mathtt{101}$.

Thus, at the end of the second iteration, the points-to sets are

$\mathtt{x} \rightarrow \{\}, \mathtt{y} \rightarrow \{\}, \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}, \mathtt{q} \rightarrow \{\mathtt{a}\}$.

Executing the final step of evaluating the generative constraints, we obtain two additional linear constraints: $\mathtt{x} = \mathtt{b}, \mathtt{y} = \mathtt{b}$.

Following the same process, at the end of the third iteration we get $\mathtt{x} = \mathtt{29}, \mathtt{y} = \mathtt{29}$, $\mathtt{a} = \mathtt{17} \times \mathtt{29}, \mathtt{b} = \mathtt{29}, \mathtt{c} = \mathtt{17} \times \mathtt{29}, \mathtt{p} = \mathtt{17} \times \mathtt{29} \times \mathtt{101}, \mathtt{q} = \mathtt{17} \times \mathtt{29} \times \mathtt{101}$ which corresponds to the points-to set

$\mathtt{x} \rightarrow \{\mathtt{y}\}, \mathtt{y} \rightarrow \{\mathtt{y}\}, \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}, \mathtt{q} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}$

and no new linear constraints are added.

The fourth iteration makes no change to the values of the variables suggesting that a fixpoint solution is reached.

2.3 The Algorithm

Our points-to analysis is outlined in Algorithm 1. To avoid clutter, we have removed the details of pruning of powers, which is straightforward. The analysis assumes availability of the set of constraints C and the set of variables V used in C. An important data structure is the prime-factor-table which is implemented as a hash-table mapping a key to a set of prime numbers that form the factors of the key. Insertion of the tuple ($\mathtt{a} \times \mathtt{b}$, a, b) assumes existence of a and b in the table (our analysis guarantees that) if a or b is not unity, and is done by combining the prime factors for a and $\mathtt{b} \in \mathcal{P}$ from the table.

Algorithm 1. Points-to analysis as a system of equations

```
Require: set C of points-to constraints, set V of variables
Ensure: each variable in V has a value indicating its points-to set
    for all v ∈ V do
      v = 1
    end for
    for each constraint c in C do
 5:   if c is an address-of constraint a = &b then
        address-of(b) = nextprime()
        prime-factor-table.insert(a× address-of(b), a, address-of(b))
        a = a× address-of(b);
        C.remove(c)
10:   else if c is a store constraint *a = b then
        generative-constraints.add(c)
        C.remove(c)
      else if c is a load constraint a = *b then
        c = constraint(a = b + 1)
15:   end if
    end for

    repeat
      for all v ∈ V do
20:     v_copy = v
      end for
      for all c ∈ C of the form v = e do
        renamed = defined(v)
        if renamed == 0 then
25:       c = constraint(v = v_copy × e)
        else
          c = constraint(v^renamed = v_copy × e)
          merge-constraints.add(constraint(v = v^renamed))
        end if
30:     ++defined(v)
      end for
      V = linear-solve(C)
      for all v ∈ V do
        v = interpret(v, v_copy, V, prime-factor-table) {Algo. 3}
35:   end for
      for all c ∈ merging-constraints of the form v_1 = v_2 do
        prime-factor-table.insert(v_1 × v_2, v_1, v_2)
        v_1 = v_1 × v_2
      end for
40:   for all c ∈ generative-constraints of the form *a = b do
        S = get-points-to(a, prime-factor-table) {Algo. 2}
        for all s ∈ S do
          C.add(constraint(s = b))
        end for
45:   end for
    until V == set(v_copy)
```

Algorithm 2. Finding points-to set

```
Require: Value v, prime-factor-table
1: S = {}
2: P = get-prime-factors(v, prime-factor-table)
3: for all p ∈ P do
4:   S = S∪ reverse-lvalue(p)
5: end for
6: return S
```

Algorithm 3. Interpreting values

```
Require: Value v, Value v_copy, set of variables V, prime-factor-table
    if v == 1 then
        return v
    else if v ∈ prime-factor-table then
        return v
 5: else if v/v_copy ∈ prime-factor-table then
        prime-factor-table.insert(v, v_copy, v/v_copy)
        return v
    else
        v = v/v_copy
10:     k = 1
        while (v − k) ∉ prime-factor-table do
            ++k
        end while
        v = (v − k)
15:     for i = 1 to k do
            S = get-points-to(v, prime-factor-table) {Algo. 2}
            prod = 1
            for all s ∈ S and s ≠ 1 do
                r = reverse-lvalue(s)
20:             prod = prod × r
                prime-factor-table.insert(prod, prod/r, r)
            end for
            v = prod
        end for
25: end if
    return v × v_copy
```

The important step of interpreting the solution is done in Lines 33–35 using Algorithm 3. The algorithm checks for an entry of a variable's value in the prime-factor-table to see if it is a valid composition of primes. Both Algorithms 1 and 3 make use of Algorithm 2 for computing points-to set of a pointer. It finds the prime factors of the r-value of the pointer (Line 2) followed by an unmapping from the primes to the corresponding variables (Line 4).

At the end of Algorithm 1, the r-values of variables in V denote their computed points-to sets. C is no longer required.

Implementation issue. Similar to other works on finding linear relationships among program variables [4,22], our analysis suffers from the issue of large values. Since we store points-to set as a multiplication of primes, the resulting values quickly go beyond the integer range of 64 bits. Hence we are required to use an integer library (GNU MP Bignum Library [13]) that emulates integer arithmetic over large unsigned integers.

2.4 Context-Sensitive Analysis

We extend Algorithm 1 for context-sensitivity using an invocation graph based approach [7]. It enables us to disallow non-realizable interprocedural execution paths. We handle recursion by iterating over the cyclic call-chain and computing a fixpoint of the points-to tuples. Our analysis is field-insensitive, i.e., we assume that any reference to a field inside a structure is to the whole structure.

3 Soundness and Precision

Soundness implies that our algorithm identifies every points-to fact identified by an inclusion-based analysis. Precision implies that our analysis does not compute a (proper) superset of the information compared to an inclusion-based analysis.

We first prove three properties of the solution to the system of linear equations.

Property 1: *Feasibility.*

Proof: By renaming the variable occurring in multiple assignments as $\mathtt{a}', \mathtt{a}'', ...$, we guarantee at most one definition per variable. Further, all constants involved in the equations are positive. Thus, there is no negative weight cycle in the constraint graph — in fact, there is neither a cycle nor a negative weight. This guarantees a feasible solution to the system.

Property 2: *Uniqueness.*

Proof: A variable attains a unique value if it is defined exactly once. Our initialization of all the variables to the value of $\chi = \mathtt{1}$ followed by the variable renaming assigns a unique value to each variable. For instance, let the system have only one constraint: $\mathtt{a} = \mathtt{b}$. In general, this system has infinite number of solutions because b is not restricted to any value. In our analysis, we initialize both (a and b) to 1.

Property 3: *Integrality.*

Proof: We are solving (and not optimizing) a system of equations that involves only addition, subtraction and multiplication over positive integers ($\mathtt{v_i}$ and constants). Further, each equation is of the form $\mathtt{v} = \mathtt{v_i} \times \mathtt{e}$ where both $\mathtt{v_i}$ and e are integral. Hence the system guarantees an integral solution.

We now prove soundness and precision of our analysis.

Lemma 1.1: *The analysis in Algorithm 1 is monotonic.*

Proof: Every address-taken variable is represented using a distinct prime number. Second, every positive integer has a unique prime factorization. Thus, our representation does not lead to a precision loss. Multiplication by an integer corresponds to including addresses represented by its prime factors. Division by an integer maps to the removal of the unique addresses represented by its prime factors. Multiplying the equations by $\mathtt{v_{copy}}$ in iteration i (Lines 25 and 27) thus ensures encompassing the points-to set computed in iteration $\mathtt{i} - \mathtt{1}$. The only division is done in Algorithm 3 (Line 9) (which is guaranteed to be without a remainder and hence no information loss), but the product is restored in Line 26. Thus, no points-to information is ever killed and we guarantee monotonicity.

Lemma 1.2: *Address-of statements are transformed safely.*

Proof: The effect of address-of statement is computed by assigning the prime number of the address-taken variable to the r-value of the destination variable (Lines 5–9 of Algorithm 1).

Lemma 1.3: *Variable renaming is sound.*

Proof: Per constraint based semantics, statements $\mathtt{a} = \mathtt{e_1}, \mathtt{a} = \mathtt{e_2}, ..., \mathtt{a} = \mathtt{e_n}$ mean $\mathtt{a} \supseteq \mathtt{e_1}, \mathtt{a} \supseteq \mathtt{e_2}, ..., \mathtt{a} \supseteq \mathtt{e_n}$ which implies $\mathtt{a} \supseteq (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$. Renaming gives $\mathtt{a}' = \mathtt{e_1}, \mathtt{a}'' = \mathtt{e_2}, ..., \mathtt{a}'^{\mathtt{n}} = \mathtt{e_n}$ adds constraints $\mathtt{a}' \supseteq \mathtt{e_1}, \mathtt{a}'' \supseteq \mathtt{e_2}, ..., \mathtt{a}'^{\mathtt{n}} \supseteq \mathtt{e_n}$ which implies $(\mathtt{a}' \cup \mathtt{a}'' \cup ... \cup \mathtt{a}'^{\mathtt{n}}) \supseteq (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$. Merging the variables as $\mathtt{a} = \mathtt{a}', \mathtt{a} = \mathtt{a}'', ..., \mathtt{a} = \mathtt{a}'^{\mathtt{n}}$ adds constraint $\mathtt{a} \supseteq (\mathtt{a}' \cup \mathtt{a}'' \cup ... \cup \mathtt{a}'^{\mathtt{n}})$. By transitivity of $\supseteq$, $\mathtt{a} \supseteq (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$. Thus, variable renaming is sound.

Corollary 1.1: *Copy statements are transformed safely.*

Lemma 1.4: *Store statements are transformed safely.*

Proof: We define a points-to fact `f` to be *realizable* by a constraint `c` if evaluation of `c` *may* result in the computation of `f`. `f` is *strictly-realizable* by `c` if for the computation of `f`, evaluation of `c` is a *must*. For the sake of contradiction, assume that there is a valid points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ that is strictly-realizable by the store constraint $\mathtt{a} = *\mathtt{p}$ and that does not get computed in our algorithm. Since the store statement, added to the generative constraint set, adds copy constraints $\mathtt{a} = \mathtt{b}, \mathtt{a} = \mathtt{c}, \mathtt{a} = \mathtt{d}, ...$ where $\mathtt{p} \rightarrow \{\mathtt{b}, \mathtt{c}, \mathtt{d}, ...\}$ at the end of an iteration after points-to information computation and interpretation is done, the contradiction means that $\mathtt{x} \notin (*\mathtt{b} \cup *\mathtt{c} \cup *\mathtt{d} \cup ...)$. This implies, $(\mathtt{x} \notin *\mathtt{b}) \wedge (\mathtt{x} \notin *\mathtt{c}) \wedge (\mathtt{x} \notin *\mathtt{d}) \wedge ....$ This suggests that the pointee `x` propagates to the pointer `a` via some other constraints, implying that the points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ is not strictly-realizable by $\mathtt{a} = *\mathtt{p}$, contradicting our hypothesis.

Lemma 1.5: *Decomposing an r-value of* `p` *into its prime factors, unmapping the addresses as the primes to the corresponding variables, and mapping the variables to their r-values corresponds to a pointer dereference* $*\mathtt{p}$*.*

Lemma 1.6: *Load statements are transformed safely.*

Proof: For a `k`-level dereference $*^{\mathtt{k}}\mathtt{v}$ in a load statement, every '*' adds 1 to `v`'s r-value. Thus, for a unique $\mathtt{v} + \mathtt{k}$, the evaluation involves `k` dereferences. Lines 15–24 of Algorithm 3 do exactly this, and by Lemmas 1.3 and 1.5, load statements compute a safe superset.

Theorem 1: *The analysis is sound with respect to an inclusion-based analysis for a dereferencing level k.*

Proof: From Lemma 1.1—1.6 and Corollary 1.1.

Lemma 2.1: *Address-of statements are transformed precisely.*

Proof: Address of every address-taken variable is represented using a distinct prime value. Further, in Lines 5–9 of Algorithm 1, for every address-of statement $\mathtt{a} = \&\mathtt{b}$, the only primes that a is multiplied with are the addresses of bs.

Lemma 2.2: *Variable renaming is precise.*
Proof: Since each variable is defined only once and by making use of Lemma 1.3 $\mathtt{a} = (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$.

Lemma 2.3: *Copy statements are transformed precisely.*
Proof: From Lemma 2.2 and since for a transformed copy statement $\mathtt{a} = \mathtt{a_{copy}} \times \mathtt{b}$, only the primes computed as the points-to set of a so far (i.e., $\mathtt{a_{copy}}$) and those of b are included. This inclusion is guaranteed to be unique due to the uniqueness of prime factorization. Thus, a does not point to any spurious variable address.

Lemma 2.4: *Store statements are transformed precisely.*
Proof: For the sake of contradiction, assume that a points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ is computed spuriously by evaluating a store constraint $\mathtt{a} = *\mathtt{p}$ in Algorithm 1. This means at least one of the following copy constraints computed the fact: $\mathtt{a} = \mathtt{b}, \mathtt{a} = \mathtt{c}, \mathtt{a} = \mathtt{d}, ...$ where $\mathtt{p} \rightarrow \{\mathtt{b}, \mathtt{c}, \mathtt{d}, ...\}$. Thus, at least one of the copy constraints is imprecise. However, Lemma 2.3 falsifies the claim.

Lemma 2.5: *Load statements are transformed precisely.*
Proof: Number of dereferences denoted by $\mathtt{v} + \mathtt{k}$ is the same as that denoted by $*^{\mathtt{k}}\mathtt{v}$. By Lemma 1.5 and 2.3 and by the observation that Algorithm 3 does not include any extra pointee in the final dereference set.

Theorem 2: *The analysis is precise with respect to an inclusion-based analysis for a dereferencing level k.*

Proof: From Lemma 2.1–2.5.

Theorem 3: *Our analysis computes the same information as an inclusion-based points-to analysis.*

Proof: Immediate from Theorems 1 and 2.

4 Experimental Evaluation

We evaluate the effectiveness of our approach using 16 SPEC C/C++ benchmarks and two large open source programs, namely *httpd* and *sendmail*. For solving equations, we use C++ language extension of CPLEX® solver from IBM ILOG toolset [17]. We compare our approach, referred to as *linear*, with the following implementations.

– *anders*: This is the base Andersen's algorithm[1] that uses a simple iterative procedure over the points-to constraints to reach a fixpoint solution. The underlying data structure used is a sorted vector of pointees per pointer. We extend it for context-sensitivity.

- *bloom*: The bloom filter method uses an approximate representation for storing both the points-to facts and the context information using a bloom filter. As this representation results in false-positives, the method is approximate and introduces some loss in precision. For our experiments, we use the *medium* configuration [23] which results in roughly 2% of precision loss for the chosen benchmarks.
- *bdd*: This is the *Lazy Cycle Detection*(LCD) algorithm implemented using Binary Decision Diagrams (BDD) from Hardekopf and Lin [14]. We extend it for context-sensitivity.

Analysis time. The analysis times in seconds required for each benchmark by different methods are given in Table 1. The analysis time is composed of reading an input points-to constraints file, applying the analysis over the constraints and computing the final points-to information as a fixpoint.

From Table 1, we observe that *anders* goes out of memory (*OOM*) for three benchmarks: *gcc*, *perlbmk* and *vortex*. For these three benchmarks *linear* obtains the points-to information in 1–3 minutes. Further comparing the analysis time of *anders*, *bdd* and *bloom* with those of *linear*, we find that *linear* takes considerably less time for most of the benchmarks. The average analysis time per benchmark is lower for *linear* by a factor of 20 when compared to *bloom* and by 30 when compared to *bdd*. Only in the case of *sendmail, mesa, twolf* and *ammp*, the analysis time of *bloom* is significantly better. It should be noted here that *bloom* has 2% precision loss in these applications [23] compared to *anders*, *bdd* and *linear*. Last, the analysis time of *linear* is 1–2 orders of magnitude smaller than *anders*, *bdd* or *bloom*, especially for large benchmarks (*gcc, perlbmk, vortex* and *eon*). We believe that the analysis time of *linear* can be further improved by taking advantage of sharing of tasks across iterations and by exploiting properties of simple linear equations in the linear solver.

Memory. Memory requirement in KB for the benchmarks is given in Table 1. *anders* goes out of memory for three benchmarks: *gcc*, *perlbmk* and *vortex*, suggesting a need for a scalable points-to analysis. *bloom*, *bdd* and *linear* successfully complete on all benchmarks. Similar to the analysis time, our approach *linear* outperforms *anders* and *bloom* in memory requirement especially for large benchmarks. The *bdd* method, which is known for its space efficiency, uses the minimum amount of memory. On an average, *linear* consumes 21MB requiring maximum 69MB for *gcc*. This is comparable to *bdd*'s average memory requirement of 12MB and maximum of 23MB for *gcc*. This small memory requirement is a key aspect that allows our method to scale better with program size.

Query time. We measured the amount of time required to answer an alias query *alias(p, q)*. The answer is a boolean value depending upon whether pointers p and q have any common pointee. *linear* uses a GCD-based algorithm to answer the query. If GCD(p, q) is 1, the pointers do not alias; otherwise, they alias. *anders* uses a sorted vector of pointees per pointer that needs to be traversed to find a common pointee. We used a set of ${}^nP_2$ queries over the set of all n

Table 1. Time(seconds) and memory(KB) required for context-sensitive analysis

Benchmark	KLOC	Time(sec)				Memory(KB)			
		anders	bloom	bdd	linear	anders	bloom	bdd	linear
gcc	222.185	OOM	10237.7	17411.208	196.62	OOM	113577	23776	68492
httpd	125.877	17.45	52.79	47.399	76.5	225513	48036	12656	27108
sendmail	113.264	5.96	25.35	117.528	84.76	197383	49455	14320	27940
perlbmk	81.442	OOM	2632.04	5879.913	101.69	OOM	54008	17628	29864
gap	71.367	144.18	152.1	330.233	89.53	97863	31786	11116	22784
vortex	67.216	OOM	1998.5	4725.745	68.32	OOM	23486	16248	18420
mesa	59.255	1.47	10.04	21.732	58.25	8261	20702	15900	18680
crafty	20.657	20.47	46.9	154.983	45.79	15986	4095	7620	16888
twolf	20.461	0.60	5.13	27.375	23.96	1594	12656	9280	15920
vpr	17.731	29.70	88.83	199.510	47.82	50210	8901	10252	10612
eon	17.679	231.17	1241.6	2391.831	106.47	385284	87814	26864	38908
ammp	13.486	1.12	15.19	54.648	19.59	5844	5746	9964	9976
parser	11.394	55.36	145.78	618.337	55.22	121588	16201	12888	14016
gzip	8.618	0.35	1.81	6.533	2.1	1447	1205	8232	11868
bzip2	4.650	0.15	1.35	4.703	1.62	519	878	7116	10244
mcf	2.414	0.11	5.04	32.049	3.4	220	1413	6192	8336
equake	1.515	0.22	1.1	4.054	0.92	161	1494	6288	12992
art	1.272	0.17	2.4	7.678	1.26	42	637	6144	9756
average	—	—	925.76	1779.75	54.66	—	26783	12360	20711

pointers in the benchmark programs. Since it simply involves a small number of number-crunching operations, *linear* outperforms *anders* (offset by the cost of emulating large integer arithmetic). We found that the average query time for *linear* is 0.85ms compared to 1.496ms for *anders*.

5 Related Work

The area of points-to analysis is rich in literature. See [16] for a survey. We mention only the most relevant related work in this section.

Points-to analysis. Most scalable algorithms proposed are based on unification [26][10]. Steensgaard[26] proposed an almost linear time algorithm that has been shown to scale to millions of lines of programs. However, precision of unification based approaches has always been an issue. Inclusion based approaches [1] that work on subsumption of points-to sets rather than a bidirectional similarity offer a better precision at the cost of theoretically cubic complexity. Several techniques [2][15][21][27] have been proposed to improve upon the original work by Andersen. [2] extracts similarity across points-to sets while [27] exploits similarity across contexts to make brilliant use of Binary Decision Diagrams to store information in a succinct manner. The idea of *bootstrapping* [18] first reduces the problem by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [26]) and later running more precise analysis on each

of the partitions. To address the analysis cost of a completely context-sensitive analysis, approximate representations were introduced to trade off precision for scalability. [5] proposed *one level flow*, [20] unified contexts, while [23] hashed contexts to alleviate the need to store complete context information. Various enhancements have also been made for the inclusion-based analyses: online cycle elimination [9] to break dependence cycles on the fly and offline variable substitution [24] to reduce the number of pointers tracked during the analysis.

Program analysis using linear algebra. An important use of linear algebra in program analysis has been to compute affine relations among program variables [22]. [4] applied abstract interpretation for discovering equality or inequality constraints among program variables. [11] proposed an SML based solver for computing a partial approximate solution for a general system of equations used in logic programs. Another area where analyses based on linear systems has been used is in finding security vulnerabilities. [12] proposed a context-sensitive light-weight analysis modeling string manipulations as a linear program to detect buffer overrun vulnerabilities. [6] presented a tool CSSV to find string manipulation errors. It converts a program written in a restricted subset of C into an integer program with assertions. A violation of an assertion signals a possible vulnerability. Recently, [8] proposed Newtonian Program Analysis as a generic method to solve iterative program analyses using Newton's method.

6 Conclusion

In this paper, we proposed a novel approach to transform a set of points-to constraints into a system of linear equations using prime factorization. We overcome the technical challenges by partitioning our inclusion-based analysis into a linear solver phase and a post-processing phase that interprets the resulting values and updates points-to information accordingly. The novel way of representing points-to information as a composition of primes allows us to keep the equations linear in every iteration. We show that our analysis is sound and precise with respect to an inclusion-based analysis for a fixed dereference level. Using a set of 16 SPEC 2000 benchmarks and two large open source programs, we show that our approach is not only feasible, but is also competitive to the state of the art solvers. More than the performance numbers reported here, the main contribution of this paper is the novel formulation of points-to analysis as a linear system based on prime factorization. In future, we would like to apply enhancements proposed for linear systems to our analysis and improve the analysis time.

References

1. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD Thesis, DIKU, University of Copenhagen (1994)
2. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: PLDI, pp. 103–114 (2003)

3. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. McGraw Hill, New York
4. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96 (1978)
5. Das, M.: Unification-based pointer analysis with directional assignments. In: PLDI, pp. 35–46 (2000)
6. Dor, N., Rodeh, M., Sagiv, M.: Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In: PLDI (2003)
7. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI, pp. 242–256 (1994)
8. Esparza, J., Kiefer, S., Michael, L.: Newtonian program analysis. In: ICALP (2008)
9. Fähndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: PLDI (1998)
10. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: PLDI (2000)
11. Fecht, C., Seidl, H.: An even faster solver for general systems of equations. In: SAS, pp. 189–204 (1996)
12. Ganapathy, V., Jha, S., Chandler, D., Melski, D., Vitek, D.: Buffer overrun detection using linear programming and static analysis. In: CCS, pp. 345–354 (2003)
13. GNU-MP-Integer-Library, http://gmplib.org/
14. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: PLDI, pp. 290–299 (2007)
15. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In: PLDI, pp. 254–263 (2001)
16. Hind, M., Pioli, A.: Which pointer analysis should i use? In: ISSTA, pp. 113–123 (2000)
17. ILOG-Toolkit, http://www.ilog.com/
18. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI, pp. 249–259 (2008)
19. Knuth, D.: The Art of Computer Programming. Seminumerical Algorithms, vol. 2. Addison-Wesley, Reading (1997)
20. Lattner, C., Lenharth, A., Adve, V.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: PLDI, pp. 278–289 (2007)
21. Lhotak, O., Hendren, L.: Scaling Java points-to analysis using spark. In: CC (2003)
22. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL, pp. 330–341 (2004)
23. Nasre, R., Rajan, K., Ramaswamy, G., Khedker, U.P.: Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 47–62. Springer, Heidelberg (2009)
24. Rountev, A., Chandra, S.: Off-line variable substitution for scaling points-to analysis. In: PLDI, pp. 47–56 (2000)
25. Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. In: PLDI, pp. 77–90 (1999)
26. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL, pp. 32–41 (1996)
27. Whaley, J., Lam, M.S.: An efficient inclusion-based points-to analysis for strictly-typed languages. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, p. 180. Springer, Heidelberg (2002)

Strictness Meets Data Flow

Tom Schrijvers[1] and Alan Mycroft[2]

[1] Dept. of Computer Science, K.U. Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium
tom.schrijvers@cs.kuleuven.be
[2] Computer Laboratory, University of Cambridge
JJ Thomson Avenue, Cambridge CB3 0FD, UK
http://www.cl.cam.ac.uk/users/am

Abstract. Properties of programs can be formulated using various techniques: dataflow analysis, abstract interpretation and type-like inference systems. This paper reconstructs strictness analysis (establishing when function parameters are evaluated in a lazy language) as a dataflow analysis by expressing the dataflow properties as an effect system. Strictness properties so expressed give a clearer operational understanding and enable a range of additional optimisations including *implicational strictness*. At first order strictness effects have the expected principality properties (best-property inference) and can be computed simply.

1 Introduction

Fosdick and Osterweil [3] introduced the notion of *path expressions* for data flow analysis. A path expression is a regular expression that summarises a program's control graph in terms of events of interest on program variables, branches, sequences and loops.

This paper adapts the idea of path expressions to strictness analysis for lazy functional languages such as Haskell [5]. In this setting, the events of interest are evaluations of (potentially) lazy values. What sets our approach apart from traditional forms of strictness analysis based on boolean functions [2,7] or projections [9], is the combination with data flow information available in path expressions.

The combination of strictness and data flow information enables two additional forms of optimisation in addition to those based on conventional strictness and absence information. Firstly, it also captures *implicational strictness* between variables: whenever variable y is evaluated, then x has already been evaluated. Secondly, the path information reveals whether particular optimisations would apply to some but not all paths. Hence, it guides inlining to expose optimisation opportunities.

Lazy functional languages only evaluate expressions when required to progress the computation. This is similar to call-by-name in Algol60 or *normal order evaluation* in the lambda-calculus but with the additional 'laziness' requirement that repeated requests to evaluate the same expression only evaluate it once

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 439–454, 2010.
© Springer-Verlag Berlin Heidelberg 2010

and make its value available immediately to subsequent requests. The standard implementation of a value is therefore a pointer to a *thunk*; multiple references to the same value become copies of this pointer. The thunk has two states: an unevaluated state (in which the *payload* is a pointer to code to compute the value and change the thunk's state) and an evaluated state in which the payload holds the value. GHC represents the is-evaluated flag by one of two code pointers; before evaluation the flag is the thunk (which does then not need storing in the payload, and which stores its result in payload offset zero), afterwards it is a simple "load payload offset zero" code sequence. Causing a thunk to move into evaluated state is called *forcing* it.

There are two costs borne by lazy languages which their eager counterparts do not pay. Firstly, a thunk which is created but inevitably later evaluated is pointless waste of resources. Classical strictness optimisation detects this at compile time, typically to create a pre-evaluated thunk when the expression-to-be-suspended appears and to optimise away the is-evaluated test at references to the value. (*Unboxing* optimisations remove the heap allocation too.) Secondly, the is-evaluated tests on thunks are repeated on repeated references to a value. For a single variable these can often be removed at control flow points which are dominated by a *force* operation, but a contribution of this work is that this can be generalised to consider dependencies between the evaluation state of two different variables—we call this 'implicational strictness'.

2 Type-and-Effect System

Source Language. We consider a first-order functional language (see Figure 1), where a program p consists of a sequence of potentially recursive function definitions $f(x_1, \ldots, x_k) = e$. Expressions e are variables x, function application $f(e_1, \ldots, e_k)$, integer (natural number) literals n, constructor application $\texttt{succ}(e)$ and case elimination $\texttt{case}(e_1, e_2, x \rightarrow e_3)$ (where either e_2 is returned if e_1 evaluates to 0, or e_3 is returned if e_1 evaluates to $\texttt{succ}(x)$).

Types and Effects. Figure 1 lists the syntax for types and effects. Value types τ consist so far only of the type $\texttt{Nat}$ of naturals;[1] function types are of the form $\tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$ where τ_i are the argument types, τ the return type, and ϕ its effect.

An effect ϕ is either a *parameter* x_i denoting the effect of evaluating the ith function argument x_i (variables bound by `case` are effectively eager), the constant 0 for non-terminating programs, the constant 1 for effect-free programs, the sequential composition of effects $\phi_1 \cdot \phi_2$ and non-deterministic choice of effects $\phi_1 + \phi_2$. By abuse of notation, a name x_i denotes both a source-level variable and its associated effect.

[1] This means that, not counting effects, all variables and functions have exactly one type, so we do not need to introduce polymorphic types to discuss the principality of inference for types containing effects.

Source Language				**Types and Effects**		
programs	p	$::=$	$d_1 \cdots d_m$	effects	ϕ, ψ	$::= x_i$
definitions	d	$::=$	$f(x_1, \ldots, x_k) = e$			$\mid 0$
expressions	e	$::=$	x			$\mid 1$
		$\mid$	$f(e_1, \ldots, e_k)$			$\mid \phi_1 \cdot \phi_2$
		$\mid$	n			$\mid \phi_1 + \phi_2$
		$\mid$	$\mathtt{succ}(e)$	value types	τ	$::= \mathtt{Nat}$
		$\mid$	$\mathtt{case}(e_1, e_2, x \rightarrow e_3)$	function types	σ	$::= \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$

$\phi_1 + \phi_2 \equiv \phi_2 + \phi_1$	(1)	$\phi \cdot 0 \equiv 0$	(7)
$(\phi_1 + \phi_2) + \phi_3 \equiv \phi_1 + (\phi_2 + \phi_3)$	(2)	$\phi \cdot 1 \equiv \phi$	(8)
$(\phi_1 \cdot \phi_2) \cdot \phi_3 \equiv \phi_1 \cdot (\phi_2 \cdot \phi_3)$	(3)	$1 \cdot \phi \equiv \phi$	(9)
$\phi + \phi \equiv \phi$	(4)	$\phi_3 \cdot (\phi_1 + \phi_2) \equiv \phi_3 \cdot \phi_1 + \phi_3 \cdot \phi_2$	(10)
$\phi + 0 \equiv \phi$	(5)	$(\phi_1 + \phi_2) \cdot \phi_3 \equiv \phi_1 \cdot \phi_3 + \phi_2 \cdot \phi_3$	(11)
$0 \cdot \phi \equiv 0$	(6)	$x \cdot \phi \cdot x \equiv x \cdot \phi$	(12)

Fig. 1. Syntax and Equivalence Laws

Operational Semantics. Figure 2 lists the small-step operational semantics of our language, inspired by Launchbury's big-step semantics [6] for lazy evaluation. The judgement $\rho; e \overset{\phi}{\rightarrowtail} \rho'; e'$ denotes a small step from expression e and environment ρ to expression e' and environment ρ'. Values n do not reduce. An environment ρ is a map from variables to unevaluated expressions (denoted $x \mapsto e$) or evaluated values (denoted $x = n$). An expression e is an *evaluated* natural number n in ρ, denoted $e \overset{\rho}{=} n$, iff e is n, or e is a variable x and $(x = n) \in \rho$; otherwise e is *unevaluated* in ρ, denoted $e \overset{\rho}{\neq}$.

Rule (Var1) evaluates one step of unevaluated variable x, while rule (Var2) recognises that x has been fully evaluated and issues effect x. Subsequent occurrences of x are handled by rule (Var3). Rule (App) for function call is noteworthy: it replaces the call by the function body, and updates the environment with mappings from the formal arguments to the actual arguments. Following Launchbury, we assume that a renamed-apart copy of the function definition (including internal `case` bindings) is used to avoid name capture.

Not listed in the figure is the usual context rule, with context $\mathbb{C} ::= \mathtt{succ}(\cdot) \mid \mathtt{case}(\cdot, e_2, x \rightarrow e_3)$:

$$(\textsc{Context})\ \frac{\rho; e \overset{\phi}{\rightarrowtail} \rho'; e'}{\rho; \mathbb{C}[e] \overset{\phi}{\rightarrowtail} \rho'; \mathbb{C}[e']}$$

The trace ϕ of a single step is either 1 or an x (for some variable x). The transitive closure judgement $\rho; e \overset{\phi}{\rightarrowtail}_* \rho'; e'$ sequences the effects $\phi_1, \ldots, \phi_n$ of its constituent steps into a string $\phi_1 \cdot \ldots \cdot \phi_n$. Note that 0 and + effects only arise

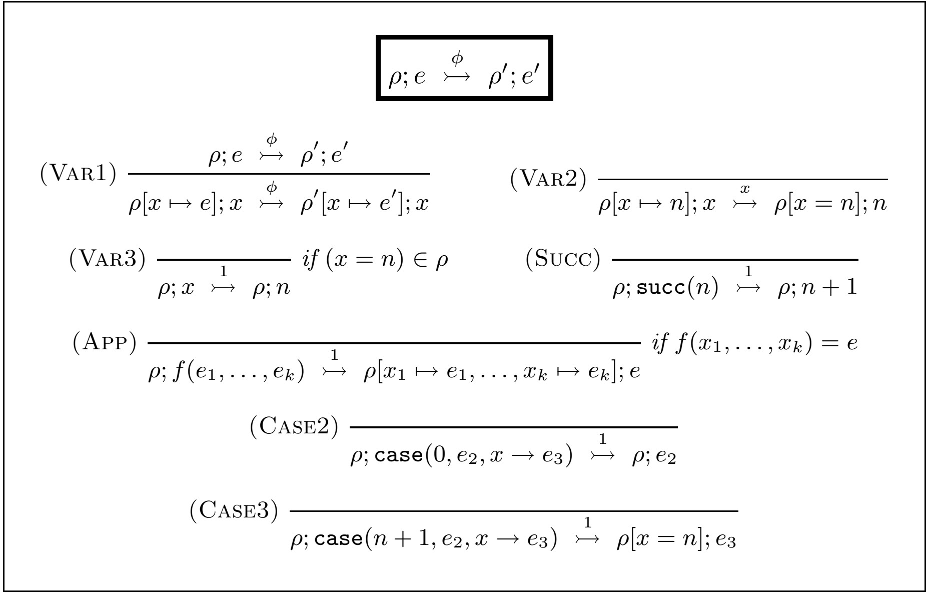


Fig. 2. Small-Step Operational Semantics

in the type-and-effect system, not through evaluation. Because the semantics does not *garbage collect* the local variables introduced by rule (App), we write $\phi|_{\{x_1,\ldots,x_n\}}$ to project ϕ onto a set of variables of interest $\{x_1, \ldots, x_n\}$. This is needed to express effect system soundness (Section 2.4).

Effect Algebra. In addition to syntactic equivalence, equivalence of effects is governed by a number of laws, listed in Figure 1. These are the forms and equality laws for regular languages (operators $+$ and $\cdot$ with units 0 and 1 and with $\cdot$ distributing over $+$) over alphabet $\{x_1, \ldots, x_k\}$, but with the additional equation (12) expressing the fact that repeated elements later (but not earlier) in a sequence are redundant.

This last law is motivated by the meaning of the parameters: x_i denotes that x_i is evaluated *at the latest* at this point. Once the effect has taken place, x_i is definitely in evaluated form. The conservative approximation lies in the fact whether x_i is evaluated at this point, or has already been evaluated before. Hence, in $x_i \cdot x_i$ we know that x_i is evaluated at the latest at the first occurrence of x_i. The second occurrence is thus redundant, because we know that x_i is already evaluated before it. In summary, we conclude that $x_i \cdot x_i \equiv x_i$.

Definition 1 (Disjunctive Normal Form). *The Disjunctive Normal Form ϕ_n of any effect ϕ is the effect obtained after exhaustive rewriting with the AC rewrite system comprised of the equivalence laws (4)–(12) interpreted as left-to-right rewrite rules. We also denote the DNF of ϕ as $dnf(\phi)$.*

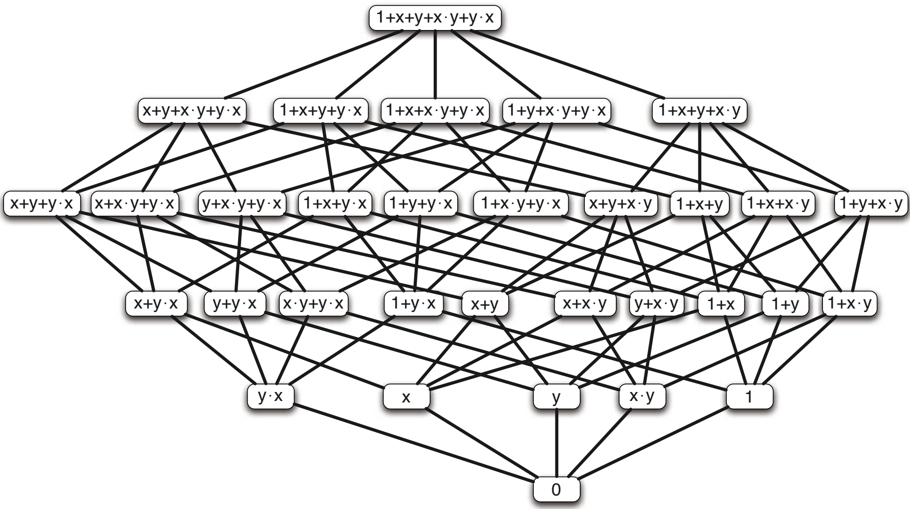


Fig. 3. The 32 different effects involving the two variables x and y

The Disjunctive Normal Form (DNF) is a non-deterministic choice of sequential compositions. Each effect has a DNF that is unique modulo associativity and commutativity. All equivalent effects have the same DNF.

Number of Distinct Effects. The number of distinct effects over a finite set of parameters is finite. For instance, the set of different effects over two parameters contains 32 elements (see Figure 3). The lines in the figure denote the "subeffect" relation, which is explained later.

The basic building blocks for effects are all permutations of k variables with $0 \leq k \leq n$; there are $\sum_{k=0}^{n} \frac{n!}{k!}$ such building blocks for n variables. Note that the permutation of length 0 denotes the effect 1. For instance, for $n = 1$ there are 2 building blocks: 1 and x_1. For $n = 2$ there are 5: 1, x_1, x_2, $x_1 \cdot x_2$ and $x_2 \cdot x_1$.

These building blocks are combined into effects with the + operator; this yields $2^{\sum_{k=0}^{n} \frac{n!}{k!}}$ distinct effect terms that range over n parameters. Note that if none of the building blocks is selected, we obtain the effect 0. For instance, for $n = 1$ there are 4 distinct effects, and for $n = 2$ there are 32 distinct effects.

Definition 2 (Chaos). *We define the chaos effect X^* ranging over a set of parameters $X = \{x_1, \ldots, x_n\}$ as*

$$X^* = \underbrace{(1 + x_1 + \ldots x_n) \cdot \ \ldots \cdot (1 + x_1 + \ldots x_n)}_{n \ \ times}$$

Bitvector Representation. The observation about the composition of effects from building blocks suggests a bitvector representation $\bar{b}$ for effects where bit b_i denotes whether the ith building block is present or not. The ordering of building

x_1	1	ϕ
0	0	0
0	1	1
1	0	x_1
1	1	$x_1 + 1$

Fig. 4. The effect domain ranging over a single effect variable x_1

blocks in the bitvector representation may be chosen arbitrarily. Figure 4 lists the distinct effects for $n = 1$ with their bitvector representation.

Subeffects. Effects have a natural (partial) ordering—the subeffect ordering.

Definition 3. *The subeffecting relation $<:$ is the minimal relation that satisfies (up to $\equiv$) the following axiom:*

$$\phi_1 <: \phi_1 + \phi_2$$

We say that ϕ_1 is a subeffect of $\phi_1 + \phi_2$.

Note that this relation is indeed a partial order. For instance, the reflexivity property $\phi_1 <: \phi_1$ follows from choosing $\phi_2 \equiv 0$. The minimal element is 0 and the maximal element is chaos X^*. The subeffect lattice for a single variable x_1 is shown in Figure 4. The least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operators on this lattice are defined in the usual manner. Observe that they correspond to bitwise *or* $\vee$ and bitwise *and* $\wedge$ on the bit vector representation.

The $<:$ relation and $\sqcap$ and $\sqcup$ operations are lifted pointwise to function types:

$$\begin{aligned} \bar{\tau} \xrightarrow{\phi_1} \tau <: \bar{\tau} \xrightarrow{\phi_2} \tau \quad &\text{iff} \quad \phi_1 <: \phi_2 \\ (\bar{\tau} \xrightarrow{\phi_1} \tau) \sqcap (\bar{\tau} \xrightarrow{\phi_2} \tau) \quad &= \quad \bar{\tau} \xrightarrow{\phi_1 \sqcap \phi_2} \tau \end{aligned}$$

and (later) to environments Γ.

2.1 Type-and-Effect Inference System

The expression typing judgement is of the form $\Gamma \vdash e : \tau \mathbin{\&} \phi$, and denotes that expression e has type τ and its evaluation has effect ϕ with respect to the type environment Γ. In the first-order language there are separate syntactic variable names for values (x) and functions (f). Type assumptions Γ contain constraints such as $x : \tau \mathbin{\&} \phi$ and $f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$.

Figure 5 lists the rules for the type-and-effect system. Rule (Var) looks up the type of a function argument in the type environment and returns the effect corresponding to that argument. Rule (App) makes sure that the types of the arguments match the function typing in the environment.

Rules (Lit) and (Succ) cover the predefined constants. Note that to model standard implementation of arithmetic, the `succ` data constructor is strict in its argument e: the effect of evaluating `succ`(e) is the effect of evaluating e.

$$\boxed{\Gamma \vdash e : \tau \,\&\, \phi} \qquad (\textsc{Var})\ \frac{}{\Gamma \vdash x : \tau \,\&\, \phi}\ \textit{if } (x : \tau \,\&\, \phi) \in \Gamma$$

$$(\textsc{Lit})\ \frac{}{\Gamma \vdash n : \texttt{Nat} \,\&\, 1} \qquad (\textsc{Succ})\ \frac{\Gamma \vdash e : \texttt{Nat} \,\&\, \phi}{\Gamma \vdash \texttt{succ}(e) : \texttt{Nat} \,\&\, \phi}$$

$$(\textsc{App})\ \frac{\Gamma \vdash e_i : \tau_i \,\&\, \phi_i \quad (i \in 1..k)}{\Gamma \vdash f(e_1, \ldots, e_k) : \tau \,\&\, \phi[\overline{\phi_i / x_i}]}\ \textit{if } (f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau) \in \Gamma$$

$$(\textsc{Case})\ \frac{\Gamma \vdash e_1 : \texttt{Nat} \,\&\, \phi_1 \qquad \Gamma \vdash e_2 : \tau \,\&\, \phi_2 \qquad \Gamma[x : \texttt{Nat} \,\&\, 1] \vdash e_3 : \tau \,\&\, \phi_3}{\Gamma \vdash \texttt{case}(e_1, e_2, x \to e_3) : \tau \,\&\, \phi_1 \cdot (\phi_2 + \phi_3)}$$

$$\boxed{\Gamma \vdash f(\bar{x}) = e} \qquad (\textsc{Def})\ \frac{\Gamma[\bar{x} : \bar{\tau} \,\&\, \bar{x}] \vdash e : \tau \,\&\, \phi}{\Gamma \vdash f(\bar{x}) = e}\ \textit{if } (f : \bar{\tau} \xrightarrow{\phi} \tau) \in \Gamma$$

$$\boxed{\Gamma \vdash \bar{d}} \qquad (\textsc{Prog})\ \frac{\Gamma \vdash d_1 \quad \cdots \quad \Gamma \vdash d_n}{\Gamma \vdash d_1 \ \cdots \ d_n}$$

Fig. 5. Type-and-Effect Inference Rules

A function definition $f(\bar{x}) = e$ is well-typed w.r.t. environment Γ, denoted $\Gamma \vdash f(\bar{x}) = e$, if the function's typing is recorded in the environment and the function's body is well-typed w.r.t. that typing (Rule (Def)). A program $\bar{d}$ is well-typed w.r.t. environment Γ, denoted $\Gamma \vdash \bar{d}$, if all its definitions are well-typed (Rule (Prog)).

2.2 Principality

Theorem 1 (Unique Non-Recursive Function Typing). *For any Γ, there is* at most *one typing $f : \bar{\tau} \xrightarrow{\phi} \tau$ such that $\Gamma, f : \bar{\tau} \xrightarrow{\phi} \tau \vdash f(\bar{x}) = e$, if f is not recursive, i.e., e does not contain a call $f(\bar{e})$.*

Note that due to our restricted setting with only one type `Nat` there is in fact *exactly* one such function typing.

Recursive functions admit multiple typings that differ in their effect. For instance, $f(x_1) = f(x_1)$ admits typings $f : \texttt{Nat} \xrightarrow{\phi} \texttt{Nat}$ for any effect ϕ. Similarly, $f(x_1, x_2) = \texttt{case}(x_1, x_2, y \to f(y, x_2))$ has well-typings $x_1 \cdot (x_2 + \phi)$ for any ϕ. The cause of these multiple typings is the (Def) rule, which defines a recursive function's well-typing in terms of itself, i.e., as a fixpoint. Any fixpoint is a valid solution. This issue of self-reference also exists in traditional dataflow analysis. Usually, in that context, the analysis domain naturally has a lattice structure and the least (sometimes greatest) fixpoint in that lattice is the preferred one. We follow the same approach.

If two different well-typings are possible, then their greatest lower bound is also a well-typing.

Lemma 1. *For all environments Γ_1, Γ_2 and programs $\bar{d}$, if $\Gamma_1 \vdash \bar{d}$ and $\Gamma_2 \vdash \bar{d}$, then $\Gamma_1 \sqcap \Gamma_2 \vdash \bar{d}$.*

As the lattice is finite, it follows that there is a unique minimal well-typing: the principal type.

Corollary 1 (Principality). *For all environments Γ_1, Γ_2 and programs $\bar{d}$, if $\Gamma_1 \vdash \bar{d}$ and $\Gamma_2 \vdash \bar{d}$, then there exists an environment Γ such that $\Gamma <: \Gamma_1$ and $\Gamma <: \Gamma_2$ and $\Gamma \vdash \bar{d}$.*

In contrast to the data-flow-analysis approach that we follow, effect systems typically have a coercion rule:

$$(\textsc{Coerce})\ \frac{\Gamma, e : \tau_1 \,\&\, \phi_1}{\Gamma \vdash e : \tau_2 \,\&\, \phi_2}\ \textit{if}\ \tau_1 <: \tau_2\ \textit{and}\ \phi_1 <: \phi_2$$

but this is unnecessary here because (i) effects can express non-deterministic choice using $+$, and (ii) in the first-order setting, subeffecting only applies co-variantly and thus all coercions in a judgement can be pushed to the root of the proof tree and thus merged into the $<:$ of principality.

2.3 Connection to Traditional Type Inference

The type-and-effect system we have defined has the property that type inference can be done first, followed by effect inference. Type, or type-and-effect, inference can be explained in terms of reconstructing information removed by erasure operators. Erasure of types, and reconstructing types without effects is standard. So we now consider an erasure operator which removes effects from expression types-and-effects and from function types yielding *traditional types* (which in our case are *simple types* but could equally be Hindley-Milner polymorphic types), and state various results. Effect erasure is defined as follows:

$$\epsilon(\tau \,\&\, \phi) = \epsilon(\tau) \qquad \epsilon(\texttt{Nat}) = \texttt{Nat} \qquad \epsilon(\bar{\tau} \xrightarrow{\phi} \tau) = \bar{\tau} \rightarrow \tau$$

and lifted to environments Γ as usual.

Results. A well-typing ($\vdash$) in the type-and-effect system is also a traditional well-typing ($\vdash_T$). Conversely, a well-traditional-typing always has a well-typing in the type-and-effect system (i.e. the type-and-effect system is a conservative extension).

Theorem 2 (Conservative Extension)

$$(\forall e, \Gamma, \tau) \quad \epsilon(\Gamma) \vdash_T e : \epsilon(\tau) \;\Leftrightarrow\; (\exists \phi) \;\; \Gamma \vdash e : \tau \,\&\, \phi$$

2.4 Effect System Soundness

We have presented a semantics and an effect system for our simple language and now address their consistency. To establish soundness, we show an enriched form of progress and preservation lemmas,[2] after an auxiliary definition.

Definition 4 (Compatible Environments). *A typing environment Γ is compatible with an evaluation environment ρ iff $\Gamma = tenv(\rho)$ where*

$$tenv(\rho) = \{x : \mathtt{Nat} \,\&\, x \mid (x \mapsto e) \in \rho\} \cup \{x : \mathtt{Nat} \,\&\, 1 \mid (x = n) \in \rho\}$$

Progress expresses that a well-typed non-value expression is never stuck.

Lemma 2 (Progress)

$$(\forall \rho_1, e_1, \tau, \phi_1) \quad tenv(\rho_1) \vdash e_1 : \tau \,\&\, \phi_1 \quad \Rightarrow \quad (\exists \rho_2, e_2, \phi_2) \quad \rho_1; e_1 \overset{\phi_2}{\rightarrowtail} \rho_2; e_2$$

where e_1 is not a value.

The preservation lemma expresses that the types and effects before and after an evaluation step are related appropriately: the type is the same and the original effect subsumes the concatenation of the evaluation trace and the new effect.

Lemma 3 (Preservation)

$$\begin{aligned}(\forall \rho_1, \rho_2, e_1, e_2, \tau, \phi_1, \phi_2) \quad & tenv(\rho_1) \vdash e_1 : \tau \,\&\, \phi_1 \quad \wedge \quad \rho_1; e_1 \overset{\phi_2}{\rightarrowtail} \rho_2; e_2 \\ \Rightarrow \quad (\exists \phi_3) \quad & tenv(\rho_2) \vdash e_2 : \tau \,\&\, \phi_3 \quad \wedge \quad (\phi_2 \cdot \phi_3)|_{dom(\rho_1)} <: \phi_1\end{aligned}$$

3 Inference Algorithm

Figure 6 lists our first-order inference algorithm. The inference judgement for expressions is of the form $\Gamma \vdash^{\mathcal{A}} e : \tau \,\&\, \phi \mid C$, which denotes that type τ and effect ϕ are inferred for expression e with respect to environment Γ and with Γ and τ subject to a set C of type equality constraints of the form $\tau = \tau'$.[3]

The type-and-effect information in the inference algorithm are essentially independent. The type-related part of the algorithm corresponds to traditional type inference as discussed earlier.

The effect inference for expressions is fairly straightforward. A composite expression's effect is a composite effect, composed from the components' effects. Note that in each case the minimal effect of an expression is returned.

The hardest part of effect inference takes place for a function definition. For recursive calls during the inference of the function body, we use a meta-effect γ as a place-holder. The body's inference returns an effect ϕ for the function that

[2] The traditional lemmas are recovered through effect erasure.

[3] $\models C$ denotes that C is satisfiable, usually established by unification.

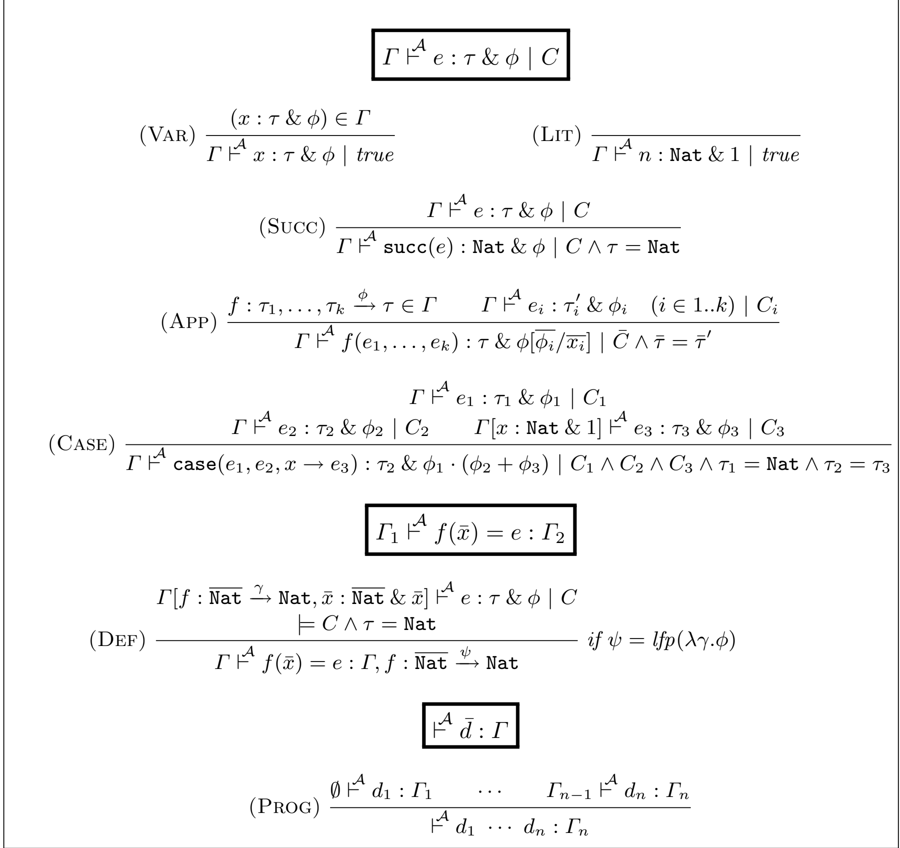


Fig. 6. Syntax-Directed Inference Algorithm

potentially mentions γ. In order to obtain a proper effect for the function, the equation $\phi <: \gamma$ must be solved. The least solutions of this inequation is obtained as the least fixpoint of $\mu\gamma.\phi$, starting from 0. The number of iterations needed to obtain the least fixpoint is bounded from above by the number of distinct variable permutations, but may be much smaller in practice.

Example 1. Consider the function definition $g(x_1, x_2) = \mathtt{case}(x_1, x_2, y \to g(y, x_2))$ with effect equation $x_1 \cdot (x_2 + \phi[1/x_1, x_2/x_2]) <: \phi$. We obtain the least fixpoint in two steps, and confirm it in the third step:

$$\begin{aligned}
\phi_0 &\equiv 0 \\
\phi_1 &\equiv x_1 \cdot (x_2 + \phi_0[1/x_1, x_2/x_2]) \equiv x_1 \cdot x_2 \\
\phi_2 &\equiv x_1 \cdot (x_2 + \phi_1[1/x_1, x_2/x_2]) \equiv x_1 \cdot x_2
\end{aligned}$$

3.1 Properties

Theorem 3 (Soundness & Completeness wrt. the Inference System). *If* $\vdash^{\mathcal{A}} \bar{d} : \Gamma$*, then* $\Gamma \vdash \bar{d}$*, for any program* $\bar{d}$ *and environment* Γ*. If* $\Gamma \vdash \bar{d}$*, then* $\vdash^{\mathcal{A}} \bar{d} : \Gamma'$*, for any program* $\bar{d}$ *and environment* Γ *and for some* Γ'*.*

Theorem 4 (Principality). *If* $\Gamma \vdash \bar{d}$ *and* $\vdash^{\mathcal{A}} \bar{d} : \Gamma'$*, then* $\Gamma' <: \Gamma$ *for any program* $\bar{d}$ *and environments* Γ, Γ'*.*

Theorem 5 (Termination). *The inference algorithm terminates for any program* $\bar{d}$*.*

4 Optimisations

A number of different optimisations are possible.

4.1 Standard Strictness Analysis and Optimisations

Our strictness domain is more expressive than the Boolean expressions used in traditional strictness analysis. The abstraction relation α maps our effects to Boolean expressions.

$$\begin{array}{lll} \alpha(1) = 1 & \alpha(\phi_1 \cdot \phi_2) \;= \alpha(\phi_1) \wedge \alpha(\phi_2) & \alpha(x_i) = x_i \\ \alpha(0) = 0 & \alpha(\phi_1 + \phi_2) = \alpha(\phi_1) \vee \alpha(\phi_2) & \end{array}$$

Moreover $\alpha(\phi[\phi'/x]) = \alpha(\phi)[\alpha(\phi')/x]$.

Lemma 4 (Well-definedness). *Equivalent effects abstract to equivalent boolean functions:*

$$(\forall \phi_1, \phi_2) \quad \phi_1 \equiv \phi_2 \quad \Rightarrow \quad \alpha(\phi_1) \equiv \alpha(\phi_2)$$

The converse does not hold. Consider $\phi_1 = 1 + x_1$ and $\phi_2 = 1$. While $\phi_1 \not\equiv \phi_2$, we do have that $\alpha(\phi_1) \equiv \alpha(\phi_2) \equiv 1$. Hence, the α mapping is an abstraction because it loses information.

Theorem 6 (Complete abstraction). *Given program* $\bar{d}$ *and writing* $\vdash^{\mathcal{S}}$ *for standard boolean strictness inference using boolean functions, then* $\vdash^{\mathcal{S}} \bar{d} : \alpha(\Gamma) \Leftrightarrow \vdash^{\mathcal{A}} \bar{d} : \Gamma$*.*

The following two optimisations are enabled by standard strictness analysis.

Eager Evaluation. If a function is strict in an argument, then that argument may be evaluated before the function call. A function is strict in argument x_i if $\alpha(\phi[0/x_i]) \equiv 0$. Since 0 is the only effect ϕ for which $\alpha(\phi) = 0$, we can equally check argument strictness by testing $\phi[0/x_i] \equiv 0$.

Loop Detection. As in traditional strictness, if an expression e has effect 0, then its evaluation does not terminate. Hence, it may be replaced by `loop()`:

- If `loop` is defined as `loop() = loop()`, the transformed code should run in constant space, whereas e may not.
- Alternatively, defining `loop() = error("loop!")`, using a Haskell feature, transforms the code to abort evaluation and report non-termination to the programmer.

4.2 Inlining to Expose Standard Strictness Optimisation

If a function is not strict in an argument, standard strictness optimisations do not apply. However, not being strict in an argument may mean either that the function never evaluates its argument or only sometimes evaluates it. In the latter case, there are one or more branches that do not evaluate the argument and one or more that do evaluate it. Inlining and floating the actual arguments into the branches, may effectively enable standard strictness optimisations. Our effects can be useful for guiding inlining.

For instance, if $f(x_1) = e$ has effect 1, this means that inlining of f will not uncover any opportunities for strictness optimisation, while $1 + x_1$ promises opportunities for parameter x_1.

Example 2. The function $f(x_1, x_2) = \texttt{case}(x_1, 0, y \to x_2)$ has type

$$f : \texttt{Nat}, \texttt{Nat} \xrightarrow{x_1 \cdot (1+x_2)} \texttt{Nat}$$

which provides no direct opportunity for strictness optimisation of x_2. However, after inlining, strictness optimisation can be applied to the second branch of f.

Note that in general inlining of a single function f may not be sufficient to uncover optimisation opportunities. Take f to be defined as $f(x_1) = g(x_1)$ where g has the effect $1 + x_1$ to illustrate this point. In the worse case, we may need to inline successively all the functions in the program to expose a strictness optimisation opportunity guaranteed by the typing.

4.3 Absent Argument Optimisation

If a function does not use (i.e. evaluate) its argument, then the argument is effectively dead code. So instead of the actual argument, the caller may provide a dummy argument or even no argument at all, i.e. an absent argument [9].

A function of type $f : \tau_1, \ldots, \tau_k \xrightarrow{\phi} \tau$ does not evaluate its ith argument (on any path which can return) if $x_i \notin \phi$. For instance, a function of type $f : \texttt{Nat} \xrightarrow{1} \texttt{Nat}$ does not evaluate its argument. Hence, the function definition can be rewritten from $f(\ldots, x_{i-1}, x_i, x_{i+1}, \ldots) = e$ to $f(\ldots, x_{i-1}, x_{i+1}) = e$, and likewise the ith argument may be dropped from all calls in the program. It is important to do Loop Detection Optimisation first (which replaces paths, including possible references to x_i on them, which can never return with $\texttt{loop()}$), consider e.g. $f(x) = \texttt{case}(x, f(x), y \to f(x))$.

Note that absent argument information is not present in the traditional strictness domain. There we have that $\alpha(1) = 1 = \alpha(1 + x_1)$.

4.4 Implicational Strictness

A standard optimisation exploits the explicit *intraprocedural* data flow and avoids consecutive evaluation of the same variable. For instance, the second occurrence of x in $\texttt{case}(x, \texttt{case}(x, e_1, y \to e_3), z \to e_4)$, is known to have been evaluated already. So the expression can be replaced with $\texttt{case}(x, \texttt{case\#}(x, e_1, y \to$

$e_3), z \to e_4)$, where `case#` does not force its argument (i.e. reads the payload of its discriminant directly):

$$(\text{CASE\#})\ \frac{}{\rho; \texttt{case\#}(e_1, e_2, x \to e_3) \overset{1}{\rightarrowtail} \rho; \texttt{case}(n, e_2, x \to e_3)}\ \textit{if } e_1 \overset{\rho}{=} n$$

For now, we leave `case#` stuck at unevaluated expressions, but come back to this issue later. Bolingbroke and Peyton Jones [1] show how this availability optimisation is easily implemented using a straightforward common-subexpression elimination in a strict core language.

Traditionally, this optimisation does not work across procedure boundaries, because the data flow within a function definition is hidden. Our new strictness domain exposes the relative evaluation order of function arguments across procedure boundaries; this information enables the *interprocedural* form of the optimisation. Consider a function $f(x, y)$ with effect $1 + x \cdot y$; this has two returning control-flow paths, one evaluating neither variable and one evaluating y after x. While f is not strict in x or y (nor jointly strict in x and y as in arms of a conditional) we do know that, given a call $f(e_1, e_2)$, then whenever e_2 is evaluated the thunk for e_1 will already have been forced. This allows us to optimise a call $f(x, \texttt{case}(x, 0, z \to z))$, logically $f(x, x-1)$, to $f(x, \texttt{case\#}(x, 0, z \to z))$

Hence we are interested in partial order information "is-always-evaluated-before". Each effect ϕ defines a partial order $\prec_\phi$ on the set of effect variables X as follows.

Definition 5 (Variable Evaluation Order). *We say that a variable x_1 must[4] be evaluated before variable x_2 with respect to effect ϕ in DNF, denoted $x_1 \prec_\phi x_2$, iff (with $x \neq x_1, x \neq x_2$)*

$$\begin{aligned} x_1 \prec_{x\cdot\phi} x_2 &= x_1 \prec_\phi x_2 & x_1 \prec_1 x_2 &= \textit{true} \\ x_1 \prec_{x_1\cdot\phi} x_2 &= \textit{true} & x_1 \prec_0 x_2 &= \textit{true} \\ x_1 \prec_{x_2\cdot\phi} x_2 &= \textit{false} & x_1 \prec_{\phi_1+\phi_2} x_2 &= x_1 \prec_{\phi_1} x_2 \wedge x_1 \prec_{\phi_2} x_2 \end{aligned}$$

For effects ϕ that are not in DNF, the relation is defined as:

$$x_1 \prec_\phi x_2 = x_1 \prec_{dnf(\phi)} x_2$$

For instance, the effect $x_1 \cdot x_2 + x_1 \cdot x_3$ captures the following order information:

$\prec$	x_1	x_2	x_3
x_1	$-$	Y	Y
x_2	N	$-$	N
x_3	N	N	$-$

as does $x_1 \cdot (x_2 + x_3)$. Note that in the case of 0 we can choose the variable evaluation order arbitrarily.

It is important to note that $\prec_\phi$ does not respect $\equiv$ (and hence is not a congruence for terms not in DNF), due to the behaviour of 0. For example, suppose

[4] Only paths which can terminate are considered.

we have code f with one path which evaluates first x and then y. This has effect $\phi = x \cdot y$ and so $x \prec_\phi y$ holds, but not $y \prec_\phi x$. Suppose now there is a definite loop before, or more problematically after, this code. Now its effect is $\phi' = 0 = \phi \cdot 0 = 0 \cdot \phi$ and note that both $x \prec_{\phi'} y$ and $y \prec_{\phi'} x$ hold. This appears paradoxical, in that code which evaluates x first and then y and then loops can be deduced to evaluate y before x! The resolution is that only paths which can return a result are considered by the $\prec_\phi$ relation; and using an incorrect order of evaluation on non-terminating paths does not matter (save for an implementation effect we explore in the next section). This effect also occurs if the code has multiple paths; the effect of 0 is to remove guaranteed non-terminating paths from consideration in the overall effect.

4.5 Transformation Soundness

There is a subtlety concerning the path 0 which we noted above. 0 represents a path which can never return, the archetypal example being a function call `loop()` given a definition `loop() = loop()`. While 0 behaves as an identity for $+$ and a (left and right) zero for $\cdot$ these algebraic properties which are fine for analysis need care when being used for optimisation.

This is related to partial versus total correctness: given function $f(x, y)$ having effect $x \prec_\phi y$ should not be simply read as "x is always evaluated before y", but more properly should be read as "x is always evaluated before y *whenever f returns*".

While the exact behaviour of code on non-terminating paths is not in general interesting, we must be careful that data-representation errors do not occur (these could replace non-termination with memory faults, or even seemingly valid answers). Consider again optimising a call $f(x, \texttt{case}(x, 0, z \to z))$ to $f(x, \texttt{case\#}(x, 0, z \to z))$ when we know $f(x, y)$ has effect ϕ and we have $x \prec_\phi y$. The problem is that f could have a definition such as $f(x, y) = \texttt{case}(y, f(x, y), z \to f(x, y))$ which would cause the potentially unevaluated thunk for x in the second argument of the call to be discriminated by `case#`. While this is clearly not a problem for unboxed values such as small integers and booleans (since the question is which of two infinite loops are taken), for values represented as pointers to code or to data this can spell memory errors or branches to arbitrary locations.

Formally, we model the issue with (i) an erroneous effect Err, that is generated when `case#` encounters an unevaluated expression, and (ii) a non-deterministic value n.

$$(\textsc{Err}) \; \frac{}{\rho; \texttt{case\#}(e_1, e_2, x \to e_3) \overset{\textsc{Err}}{\rightarrowtail} \rho; \texttt{case}(n, e_2, x \to e_3)} \; \textit{if } e \not\Downarrow^{\rho} n \quad (\forall n)$$

A transformation that introduces `case#` necessarily (by soundness) avoids the (Err) transition on all terminating derivations (non-0 paths). Otherwise a change in semantics can be observed: the Err effect shows up in the trace of the transformed program, but not the original program. For non-terminating derivations, we distinguish between *fragile* and *robust* implementations.

Fragile implementations distinguish between the erroneous transition (e.g. yielding a crash) and non-termination. This is modelled by the supplementary law $\text{Err} \cdot 0 \not\equiv 0$ overriding the more general $\phi \cdot 0 \equiv 0$. Thus any (Err) transition, whether for a terminating or non-terminating derivation, violates the soundness of the optimisation. There are various ways to avoid (Err) on the 0 path for fragile implementations (e.g. dropping the "right zero" law), but we prefer to avoid the 0 path itself, analogous to traditional dataflow analysis, by adding a dummy return node to every loop that otherwise would not have one. The downside of avoiding the (Err) transition is of course the reduced opportunity for optimisations, which is why we turn to robust implementations.

Robust implementations are still modelled by $\text{Err}\cdot 0 \equiv 0$, and (Err) transitions in non-terminating derivations are not observable. This is for instance possible by ensuring that the payload of an unevaluated thunk is always interpretable as a value of its result type; hence the non-deterministic value n in rule (Err).

5 Related Work

Jensen [4] presents a strictness analysis based on strictness logic. His strictness language is perhaps the closest to ours, with polymorphic variables α and conditional strictness $\phi_2?\phi_1$ similar to respectively our parameters x and sequential composition $\phi_1 \cdot \phi_2$. However, as it lacks branching (+) and 0, our novel optimisations do not apply.

We have only considered *flat* data, i.e. the natural numbers, where forcing the value also forces the component. Wadler [8] shows one way to extend strictness analysis to non-flat domains. A similar technique would apply to our domain.

Wansbrough [10] annotates function types with polymorphic "usage" annotations to identify thunks which are encountered at most once; these can be optimised to remove the "is-evaluated" test. It is appealing to speculate whether an extended effect system can capture this property too.

6 Conclusion and Future Work

We have expressed strictness as effects in a type-and-effect system which both adds insight into strictness properties and provides additional strictness optimisation opportunities.

There is a natural extension of our type-and-effect system to higher-order and polymorphic types (we need effect variables and an effect binding construct so that $\lambda x.x$ has effect $\forall \alpha.\alpha \xrightarrow[x]{x} \alpha$). With a subtyping rule, similar to (Coerce) in Section 2.2, the system becomes a conservative extension of polymorphic types, however it remains unclear whether the type-and-effect system has principal types, necessary for a type inference algorithm.

Acknowledgements. Tom Schrijvers gratefully acknowledges funding for visiting the University of Cambridge from the Fund for Scientific Research – Flanders. The authors thank the anonymous reviewers for their helpful comments.

References

1. Bolingbroke, M.C., Peyton Jones, S.L.: Types are calling conventions. In: Haskell 2009: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, pp. 1–12. ACM, New York (2009)
2. Burn, G.L., Hankin, C.L., Abramsky, S.: The theory of strictness analysis for higher order functions. In: On Programs as Data Objects, New York, NY, USA, pp. 42–62. Springer, Heidelberg (1985)
3. Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. ACM Comput. Surv. 8(3), 305–330 (1976)
4. Jensen, T.P.: Inference of polymorphic and conditional strictness properties. In: POPL 1998: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 209–221. ACM, New York (1998)
5. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries – The Revised Report (2003)
6. Launchbury, J.: A natural semantics for lazy evaluation. In: POPL 1993: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 144–154. ACM, New York (1993)
7. Mycroft, A.: Abstract interpretation and Optimising Transformations for Applicative Programs. PhD thesis, University of Edinburgh (1981)
8. Wadler, P.: Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In: Abstract Interpretation. Ellis Horwood (1987)
9. Wadler, P., Hughes, R.J.M.: Projections for strictness analysis. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 385–407. Springer, Heidelberg (1987)
10. Wansbrough, K.: Simple polymorphic usage analysis. Technical Report UCAM-CL-TR-623, Cambridge University Computer Laboratory (March 2005)

Automatic Verification of Determinism for Structured Parallel Programs

Martin Vechev[1], Eran Yahav[1], Raghavan Raman[2], and Vivek Sarkar[2]

[1] IBM T.J. Watson Research Center
{mtvechev,eyahav}@us.ibm.com
[2] Rice University
{raghav,vsarkar}@rice.edu

Abstract. We present a static analysis for automatically verifying determinism of structured parallel programs. The main idea is to leverage the structure of the program to reduce determinism verification to an independence property that can be proved using a simple sequential analysis. Given a task-parallel program, we identify program fragments that may execute in parallel and check that these fragments perform independent memory accesses using a sequential analysis. Since the parts that can execute in parallel are typically only a small fraction of the program, we can employ powerful numerical abstractions to establish that tasks executing in parallel only perform independent memory accesses. We have implemented our analysis in a tool called DICE and successfully applied it to verify determinism on a suite of benchmarks derived from those used in the high-performance computing community.

1 Introduction

One of the main difficulties in parallel programming is the need to reason about possible interleavings of concurrent operations. The vast number of interleavings makes this task difficult even for small programs, and impossible for any sizeable software.

To simplify reasoning about parallel programs, it is desirable to reduce the number of interleavings that a programmer has to consider [19,4]. One way to achieve that is to require parallel programs to be *deterministic*. Informally, determinism means that for a given input state, the parallel program will always produce the same output state. Determinism is an attractive correctness property as it abstracts away the interleavings underlying a computation.

In this paper, we present a technique for automatic verification of determinism. A key feature of our approach is that it uses *sequential analysis* to establish independence of statements in the parallel program. The analysis works by applying simple assume-guarantee reasoning: the code of each task is analyzed *sequentially*, under the assumption that all memory locations the task accesses are independent from locations accessed by tasks that may execute in parallel. Then, based on the sequential proofs produced in the first phase, the analysis checks whether the independence assumption holds: for each pair of statements that may execute in parallel, it (conservatively) checks that their memory accesses are independent. Our analysis does not assume any a priori bounds on the number of heap allocated objects, the number of tasks, or sizes of arrays.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 455–471, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Our approach can be viewed as automatic application of the Owicki/Gries method, used to check independence assertions. The restricted structure of parallelism limits the code for which we have to perform interference checks and enables us to use powerful (and costly) numerical domains.

Because in our language arrays are heap allocated objects, our analysis combines information about the heap with information about array indices. We leverage advanced numerical domains such as Octagon [23] and Polyhedra [7] to establish independence of array accesses. We show that tracking the relationships between index variables in realistic parallel programs requires such rich domains.

There has been a large volume of work on establishing independence of statements in the context of automatic parallelization (e.g.,[17,2,24]). These techniques were intended to be part of a parallelizing compiler, and their emphasis is on efficiency. Hence, they usually try to detect common patterns via simple structural conditions. In contrast, our focus is on verification and we use precise (and often expensive) abstract domains.

Our work can be viewed as a case study in using numerical domains for establishing determinism in realistic parallel programs. We show that proving determinism requires abstractions that are quite precise and are able to capture linear inequalities between array indices, as well as establish that different array cells point to different objects.

We implemented our analysis in a tool called DICE based on the Soot [36] analysis framework. DICE uses the Apron [15] numerical library to provide advanced numerical domains (specifically, the octagon and polyhedra domains). Our tool takes as input a normal Java program annotated with structured parallel constructs and automatically checks whether the program is deterministic. In the case where the analysis fails to prove the program as deterministic, DICE provides a description of (abstract) shared locations that potentially lead to non-determinism.

Related Work. Recently, there has been growing interest in dynamically checking determinism [5,33]. The main weakness of such dynamic techniques is that they may miss executions where determinism violations occur. Other approaches have also explored building deterministic programs by construction, both in the language [10,4] and via dynamic mechanisms such as modifying the scheduler [8,28]. A related property that has gained much attention over the years is race-freedom (e.g., [13,27,22,34,25,27,11]). However, race-freedom and determinism are not comparable properties: a parallel program can be race-free but not deterministic or deterministic but not race-free.

Main Contributions. The main contributions of this paper are:

- We present a static analysis that can prove determinism of structured parallel programs. Our analysis works by analyzing each task *sequentially*, computing assertions using a numerical domain and checking whether the computed assertions indeed imply determinism.
- We implemented our analysis in a tool called DICE based on Soot [36] and the Apron [15] numerical library. The analysis handles Java programs augmented with structured parallel constructs.
- We evaluated DICE on a set of parallel programs that are variants of the well-known Java JGF benchmarks [9]. Our analysis managed to prove five of the eight benchmarks as deterministic.

2 Overview

In this section, we informally describe our approach with a simple Java program augmented with structured parallel constructs.

2.1 Motivating Example

Fig. 1 shows the method `update` which is a simplified and normalized code fragment from the SOR benchmark program. The SOR program uses parallel computation to apply the method of successive over-relaxation for solving a system of linear equations. For this program, we would like to establish that it is deterministic.

```
1  void update(final double[][] G, final int start, final int last,
2   final double c1, final double c2, final int nm, final int ps) {
3    finish foreach (tid: [start,last]) {
4     int i = 2 * tid - ps;
5     double[] Gi = G[i];          {R:({A_G}, {idx = 2*tid - ps}) }
6     double[] Gim1 = G[i - 1];    {R:({A_G}, {idx = 2*tid - ps - 1}) }
7     double[] Gip1 = G[i + 1];    {R:({A_G}, {idx = 2*tid - ps + 1}) }
8     for (int j=1; j<nm; j++)
9      double tmp1 = Gim1[j]        {R:({A_Gim1}, {1 ≤ idx < nm})}
10     double tmp2 = Gip1[j]        {R:({A_Gip1}, {1 ≤ idx < nm})}
11     double tmp3 = Gi[j-1]        {R:({A_Gi}, {0 ≤ idx < nm - 1})}
12     double tmp4 = Gi[j+1]        {R:({A_Gi}, {2 ≤ idx < nm + 1})}
13     double tmp5 = Gi[j];         {R:({A_Gi}, {1 ≤ idx < nm})}
14     Gi[j] =                      {W:({A_Gi}, {1 ≤ idx < nm})}
15       c1 * (tmp1 + tmp2 + tmp3 + tmp4) + c2 * tmp5
16   }
17 }
```

Fig. 1. Example (normalized) code extracted from the SOR benchmark

This program is written in Java with structured parallel constructs. The **foreach (var : [l,h])** statement spawns child tasks in a loop, iterating on the value range between l and h. Each loop iteration spawns a separate task and passes a unique value in the range $[l, h]$ to that task via the task local variable `var`. A similar construct, called **invokeAll**, is available in the latest Java Fork-Join framework [18].

In addition to **foreach**, our language also supports constructs such as **fork**, **join**, **async** and **finish**. These are basic constructs with similar counterparts in languages such as X10 [6], Cilk [3] and the Java Fork-Join framework [18]. The semantics of **finish** $\{ s \}$ statement is that the task executing the finish must block and wait at the end of this statement until all descendant tasks created by this task in s (including their recursively created children tasks), have terminated.

In Fig. 1, tasks are created by **foreach** in the range of $[start, last]$. Each task spawned by the foreach loop is given a unique value for `tid`. This value is then used to compute an index for accessing a two-dimensional array `G[][]`. Because **foreach** is preceded by the **finish** construct, the main task which invoked the foreach statement cannot proceed until all concurrently executing tasks created by foreach have terminated.

Limitations. Our analysis currently does not handle usage of synchronization constructs such as monitors, locks or atomic sections.

2.2 Establishing Determinism by Independence

Our analysis is able to automatically verify determinism of this example by showing that statements that may execute in parallel either access disjoint locations or read from the same location. Our approach operates in two steps: (i) analyzing each task sequentially and computing an over-approximation of its memory accesses; (ii) checking independence of memory accesses that may execute in parallel.

Computing an Over-approximation of Memory Accesses. The first step in our approach is to compute an over-approximation of the memory locations read/written by every task at a given program point. To simplify presentation, we focus on the treatment of array accesses. The treatment of object fields is similar and simpler and while we do not present the details here, our analysis also handles that case.

In our programming language, arrays are heap-allocated objects: to capture information about what array locations are accessed, our abstract representation combines information about the heap with information about array indices.

Fig. 2 shows the array `G[][]` of our running example where three tasks with task identifiers (`tid`) 1,2, and 3 access the array. In the figure, we subscript local variables with the task identifier of the task to which they belong. Note that the only columns being written to are Gi_1, Gi_2, Gi_3. Columns which are accessed by two tasks are always only read, not written. The 2D array is represented using one-dimensional arrays.

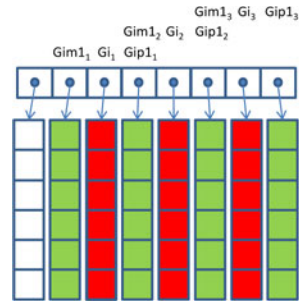


Fig. 2. Example of the array `G[][]` in SOR with three tasks with tids 1,2,3 accessing it

A key aspect of our approach is that we use simple assume/guarantee reasoning to analyze each task separately, via sequential analysis. That is, we compute an over-approximation of accessed memory locations for a task assuming that all other tasks that may execute in parallel only perform independent accesses.

Fig. 1 shows the results of our sequential analysis computing symbolic ranges for array accesses. For every program label in which an array is being accessed, we compute a pair of heap information, and array index range. The heap information records what abstract locations may be pointed to by the array base reference. The array index range records what indices of the array may be accessed by the statement via constraints on the `idx` variable. In this example, we used the polyhedra abstract domain to abstract numerical values, and the array index range is generally represented as a set of linear inequalities on local variables of the task.

For example, in line 5 of the example, the array base `G` may point to a single abstract location A_G, and the statement only reads a single cell in the array at index $2 * tid - ps$. Note that the index expression uses the task identifier tid. It is often the case in our programs that accessed array indices depend on the task identifier. Furthermore, the

coefficient for tid in this constraint is 2 and thus, this information could not have been represented directly in the Octagon numerical domain. In Section 6, we will see a variety of programs, where some can be handled by Polyhedra and some by Octagon.

Checking Independence. Next, we need to establish that array accesses of parallel tasks are independent. The only write access in this code is the write to Gi[j] in line 14. Our analysis therefore has to establish that for different values of tid (i.e., different tasks), the write in line 14 does not conflict with any of the read/write accesses made by other parallel tasks.

For example, we need to prove that when two different tasks identifiers $tid_1 \neq tid_2$ execute the write access in line 14, they will access disjoint locations. Our analysis can only do that if the pointer-analysis is precise enough to establish the fact that $G[2 * tid_1 - ps] \neq G[2 * tid_2 - ps]$ when $tid_1 \neq tid_2$. In this example program, we can indeed establish this fact automatically based on an analysis that tracks how the array G has been initialized. Generally, of course, the fact that cells of an array point to disjoint objects is hard to establish and may require expensive analyses such as shape analysis.

```
1  void update(final double[][] B, final double[][] C) {
2    finish {
3     asynch {
4      for (int i=1;i <=n; i++) {
5       double tmp1 = C[2*i];    {R:({A_C}, {2 ≤ idx ≤ 2*n})}
6       B[i] = tmp1;             {W:({A_B}, {1 ≤ idx ≤ n})}
7      }
8     }
9     asynch {
10     for (int j=n;j <=2*n; j++) {
11      double tmp2 = C[2*j+1]; {R:({A_C}, {2*n+1 ≤ idx ≤ 4*n+1})}
12      B[j] = tmp2;            {W:({A_B}, {n ≤ idx ≤ 2*n})}
13     }
14    }
15   }
16 }
```

Fig. 3. A simple example for parallel accesses to shared arrays

2.3 Reporting Potential Sources of Non-determinism

When independence of memory accesses cannot be established, our approach reports the shared memory locations that could not be established as independent.

Consider the simple example of Fig. 3. This example captures a common pattern in parallel applications where different parts of a shared array are updated in parallel. Applying our approach to this example, yields the ranges shown in the figure. Here, we used polyhedra analysis and a simple points-to analysis. Our simple points-to analysis is precise enough to establish two separate abstract locations for B and C. Checking the array ranges, however, shows that the write of line 6 overlaps with the write of line 12 on the array cell with index n. For this case, our analysis reports that the program is potentially non-deterministic due to conflicting access on the abstract locations described by $(\{A_B\}, \{\texttt{idx == n}\})$.

In some cases, such failures may be due to imprecision of the analysis. In Section 6, we discuss the abstractions required to prove determinism of several realistic programs.

3 Concrete Semantics

We assume a standard concrete semantics which defines a program state and evaluation of an expression in a program state. The semantic domains are defined in a standard way in Table 1, where *TaskIds* is a set of unique task identifiers, *VarIds* is a set of local variable identifiers, and *FieldId* is a set of (instance) field identifiers.

Table 1. Semantic Domains

$L^\natural \subseteq objs^\natural$	an unbounded set of dynamically allocated objects
$v^\natural \in Val = objs^\natural \cup \{null\} \cup \mathbb{N}$	values
$pc^\natural \in PC = TaskIds \rightharpoonup Labs$	program counters
$\rho^\natural \in Env^\natural = TaskIds \times VarIds \rightharpoonup Val$	environment
$h^\natural \in Heap^\natural = objs^\natural \times FieldId \rightharpoonup Val$	heap
$A^\natural \subseteq L^\natural$	array objects

A *program state* is a tuple: $\sigma = \langle pc^\natural_\sigma, L^\natural_\sigma, \rho^\natural_\sigma, h^\natural_\sigma, A^\natural_\sigma \rangle \in ST^\natural$, where $ST^\natural = PC \times 2^{objs^\natural} \times Env^\natural \times Heap^\natural \times 2^{objs^\natural}$.

A state σ keeps track of the program counter for each task ($pc^\natural_\sigma$), the set of allocated objects ($L^\natural_\sigma$), an environment mapping local variables to values ($\rho^\natural_\sigma$), a mapping from fields of allocated objects to values ($h^\natural_\sigma$), and a set of allocated array objects ($A^\natural_\sigma$).

We assume that program statements are labeled with unique labels. For an assignment statement at label $l \in Labs$, we denote by $lhs(l)$ the left hand side of the assignment, and by $rhs(l)$ the right hand side of the assignment.

We denote $Tasks(\sigma) = dom(pc^\natural_\sigma)$ to be the set of task identifiers in state σ, such that for each task identifier, $pc^\natural_\sigma$ assigns a value. We use $enabled(\sigma) \subseteq dom(pc^\natural_\sigma)$ to denote the set of tasks that can make a transition from σ.

3.1 Determinism

Determinism is generally defined as producing observationally equivalent outputs on all executions starting from observationally equivalent inputs.

In this paper, we establish determinism of parallel programs by proving that shared memory accesses made by statements in different tasks are independent. This is a stronger condition which sidesteps the need to define "observational equivalence", a notion that is often very challenging to define for real programs.

In the rest of the paper, we focus on the treatment of array accesses. The treatment of shared field accesses is similar (and simpler).

Definition 1 (Accessed array locations in a state). *Given a state $\sigma \in ST^\natural$, we define $W^\natural_\sigma$:* TaskIds $\rightarrow 2^{(A^\natural \times \mathbb{N})}$ *which maps a task identifier to the memory location to be*

written by the statement at label $pc_\sigma(t)$. Similarly, we define $R^\natural_\sigma$: TaskIds $\rightarrow 2^{(A^\natural \times \mathbb{N})}$ mapping a task identifier to the memory location to be read by the statement at $pc_\sigma(t)$:

$$
\begin{aligned}
&R^\natural_\sigma(t) = \{(\rho^\natural_\sigma(t,a), \rho^\natural_\sigma(t,i)) \mid pc^\natural_\sigma(t) = l \wedge rhs(l) = a[i]\} \\
&W^\natural_\sigma(t) = \{(\rho^\natural_\sigma(t,a), \rho^\natural_\sigma(t,i)) \mid pc^\natural_\sigma(t) = l \wedge lhs(l) = a[i]\} \\
&RW^\natural_\sigma(t) = R^\natural_\sigma(t) \cup W^\natural_\sigma(t)
\end{aligned}
$$

Note that $R^\natural_\sigma(t)$, $W^\natural_\sigma(t)$ and $RW^\natural_\sigma(t)$ are always singleton or empty sets.

Definition 2 (Conflicting Accesses). *Given two shared memory accesses in states $\sigma_1, \sigma_2 \in ST^\natural$, performed respectively by task identifiers t_1 and t_2, we say that the two shared accesses are* conflicting*, denoted by $(\sigma_1, t_1) \nleftrightarrow (\sigma_2, t_2)$ when: $t_1 \neq t_2$ and $W^\natural_{\sigma_1}(t_1) \cap RW^\natural_{\sigma_2}(t_2) \neq \emptyset$ or $W^\natural_{\sigma_2}(t_2) \cap RW^\natural_{\sigma_1}(t_1) \neq \emptyset$.*

Next, we define the notion of a conflicting program. A program that is not conflicting is said to be *conflict-free*.

Definition 3 (Conflicting Program). *Given the set of all reachable program states $RS \subseteq ST^\natural$, we say that the program is* conflicting *iff there exists a state $\sigma \in RS$ such that $t_1, t_2 \in Tasks(\sigma)$, $mhp(RS, \sigma, t_1, pc^\natural_\sigma(t_1), t_2, pc^\natural_\sigma(t_2)) =$ true and $(\sigma, t_1) \nleftrightarrow (\sigma, t_2)$.*

Informally, the above definition says that a program is conflicting if and only if there exists a state from which two tasks can perform memory accesses that conflict. Similar definition is provided by Shacham et. al [35]. However, our definition is more strict as we do not allow even atomic operations to conflict (recall that we currently do not handle atomic operations).

In the above definition we used the predicate $mhp : 2^{ST^\natural} \times ST^\natural \times$ *TaskIds* $\times$ *Labs* $\times$ *TaskIds* $\times$ *Labs* $\rightharpoonup Bool$. The predicate $mhp(S, \sigma, t_1, l_1, t_2, l_2)$ evaluates to *true* if t_1 and t_2 *may* run in parallel from state σ.

Computing mhp. The computation of the mhp is parametric to our analysis. That is, we can consume an mhp of arbitrary precision. For instance, we can define $mhp(S, \sigma, t_1, l_1, t_2, l_2)$ to be *true* iff $t_1, t_2 \in enabled(\sigma)$ and $t_1 \neq t_2$.

We can also define less precise (more abstract) variants of mhp. For example, $mhp(S, \sigma, t_1, l_1, t_2, l_2) =$ *true* iff $\exists \sigma' \in S, t_1, t_2 \in enabled(\sigma'), t_1 \neq t_2$ such that $pc^\natural_{\sigma'}(t_1) = l_1$ and $pc^\natural_{\sigma'}(t_2) = l_2$. As the mhp depends on S and not on σ, we can write the mhp as $mhp(S, t_1, l_1, t_2, l_2)$. This less precise definition only talks at the level of labels and may be preferable for efficiency purposes. When the set S is assumed to be all reachable programs states, we write $mhp(t_1, l_1, t_2, l_2)$.

In this paper, we use the structure of the parallel program to compute the mhp precisely, but in cases where we consider general Java programs with arbitrary concurrency, we can also use more expensive techniques [26].

3.2 Pairwise Semantics

Next, we abstract away the relationship between the different tasks and define semantics that only tracks each task separately, rather than all tasks simultaneously.

We define the projection $\sigma|_t$ of a state σ on a task identifier t as $\sigma|_t = \langle pc|_t, L, \rho|_t, h, A\rangle$, where:

- $pc|_t$ is the restriction of pc to t
- $\rho|_t$ is the restriction of ρ to t

Given a state $\sigma \in ST^\natural$, we can now define the program state for a single task t via $\sigma|_t = \langle pc, L, \rho, h, A\rangle \in \Sigma$, where $ST^\natural_{pw} = (PC \times 2^{objs^\natural} \times Env^\natural \times Heap^\natural \times 2^{objs^\natural})$. For $S \subseteq ST^\natural$:

$$\alpha_{pw}(S) = \bigcup_{\sigma \in S} \{\sigma|_t \mid t \in Tasks(\sigma)\}$$

Next, we adjust our definition of a conflicting program.

Definition 4 (Pairwise-Conflicting Program). *Given the set of all reachable program states $RS^{pw} \subseteq ST^\natural_{pw}$, we say that the program is* pairwise conflicting *when there exists $\sigma_1^{pw}, \sigma_2^{pw} \in RS^{pw}$ such that for some $t_1 \in Tasks(\sigma_1^{pw})$, $t_2 \in Tasks(\sigma_2^{pw})$, $mhp(RS^{pw}, t_1, pc^\natural_{\sigma_1^{pw}}(t_1), t_2, pc^\natural_{\sigma_2^{pw}}(t_2)) =$ true and $(\sigma_1^{pw}, t_1) \not\leftrightarrow (\sigma_2^{pw}, t_2)$.*

Note that in this definition of a conflicting program, we use Definition 2 with *two* states σ_1^{pw} and σ_2^{pw}, while in Definition 3, we used it only with a single state.

Assuming the mhp predicate computes identical results in Definition 3 and Definition 4, we now have the following simple theorem:

Theorem 1. *Any conflicting program is pairwise-conflicting.*

Of course, due to losing precision with the pairwise semantics, it could be the case that a program is pairwise-conflicting but not conflicting.

4 Abstract Semantics

The pairwise semantics tracks a potentially unbounded set of memory locations accessed by each task. In this section, we use standard abstract domains to represent sets of locations in a bounded way. We represent sets of objects using standard points-to abstractions, and ranges of array cells using numerical abstractions on array indices. Next, we abstract the semantics of Section 3.2.

4.1 Abstract State

Our abstraction is parametric on both the heap abstraction α_h and the numerical abstraction α_n. In the following, we assume an abstract numerical domain $ND = \langle NC, \sqsubseteq_{ND}\rangle$ equipped with operations $\sqcap_{ND}$ and $\sqcup_{ND}$, where NC is a set of numerical constraints over the primitive variables in *VarIds*, and do not go into further details about the particular abstract domain.

Definition 5. *An* abstract program state σ *is a tuple* $\langle pc, L_a, \rho_a, h_a, A_a, nc\rangle \in ST_a$, *where* $ST_a = PC \times 2^{objs} \times \text{Env} \times \text{Heap} \times 2^{objs} \times (TaskIds \rightarrow 2^{NC})$ *such that:*

- $L_a \subseteq$ *objs is a bounded set of abstract objects, and* $A_a \subseteq L_a$ *is a set of abstract array objects.*
- $\rho_a\colon TaskIds \times VarIds \rightarrow 2^{AVal}$ *maps a task identifier and a variable to its abstract values.*
- h_a: *objs* $\times$ FieldId $\rightarrow 2^{AVal}$ *map an abstract location and a field identifier to their abstract values.*
- $nc\colon TaskIds \rightarrow 2^{NC}$ *maps a task to a set of numerical constraints, capturing relationship between local numerical variables of that task.*

An abstract program state is a sound representation of a concrete pairwise program state $\sigma^{pw} = \langle pc^\natural, L^\natural, \rho^\natural, h^\natural, A^\natural \rangle$ when:
- $pc = pc^\natural$.
- for all $o \in L^\natural$, $\alpha_h(o) \in L_a$.
- for all $o_1, o_2 \in L^\natural$, $f \in$ *FieldId*, if $h^\natural(o_1, f) = o_2$ then $\alpha_h(o_2) \in h_a(\alpha_h(o_1), f)$.
- $dom(\rho) = dom(\rho^\natural)$
- for all task references $(t, r) \in dom(\rho^\natural)$, if $v = \rho^\natural(t, r)$ then $\alpha_h(v) \in \rho_a(t, r)$.
- Let $TL_t = \{(pr_0, v_0)...(pr_n, v_n)\}$ be the set of primitive variable-value pairs, such that for all pairs $(pr_i, v_i) \in TL_t$, $(t, pr_i) \in dom(\rho^\natural)$. Then $\alpha_n(TL_t) \sqsubseteq_{ND} nc(t)$.

Next, we define the accessed array locations in an abstract state:

Definition 6 (Accessed array locations in an abstract state). *Given an abstract state* $\sigma \in ST_a$*, we define* W_σ: TaskIds $\rightarrow 2^{(AVal \times \text{VarIds})}$ *which maps a task identifier to the memory location to be written by the statement at label* $pc_\sigma(t)$*. Similarly, we define* R_σ: TaskIds $\rightarrow 2^{(AVal \times \text{VarIds})}$ *mapping a task identifier to the memory location to be read by its statement at* $pc_\sigma(t)$*.*

$$R_\sigma(t) = \{(\rho_\sigma(t, a), i) \mid pc_\sigma(t) = l \wedge rhs(l) = a[i]\}$$
$$W_\sigma(t) = \{(\rho_\sigma(t, a), i) \mid pc_\sigma(t) = l \wedge lhs(l) = a[i]\}$$
$$RW_\sigma(t) = R_\sigma(t) \cup W_\sigma(t)$$

Note that R_σ, W_σ and RW_σ are always singleton or empty sets. We use $D_\sigma(t).B$ and $D_\sigma(t).I$ to denote the first and second components of the entry in the singleton set D, where D can be one of R, W or RW. If $D_\sigma(t)$ is empty, then $D_\sigma(t).B$ and $D_\sigma(t).I$ also return the empty set. Next, we define the notion of conflicting accesses:

Definition 7 (Abstract Conflicting Accesses). *Given two shared memory accesses in states* $\sigma_1, \sigma_2 \in ST_a$*, performed respectively by task identifiers* t_1 *and* t_2*, we say that the two shared accesses are* conflicting*, denoted by* $(\sigma_1, t_1) \nleftrightarrow_{abs} (\sigma_2, t_2)$ *when:*

- $W_{\sigma_1}(t_1).B \cap RW_{\sigma_2}(t_2).B \neq \emptyset$ *and* $(W_{\sigma_1}(t_1).I = RW_{\sigma_2}(t_2).I) \sqcap_{ND} AS \neq \bot$ *or*
- $W_{\sigma_2}(t_2).B \cap RW_{\sigma_1}(t_1).B \neq \emptyset$ *and* $(W_{\sigma_2}(t_2).I = RW_{\sigma_1}(t_1).I) \sqcap_{ND} AS \neq \bot$

where $AS = nc_{\sigma_1}(t_1) \sqcap_{ND} nc_{\sigma_2}(t_2) \sqcap_{ND} (t_1 - t_2 \geq 1)$

The definition uses the meet operation $\sqcap_{ND}$ of the underlying numerical domain to check whether the combined constraints are satisfiable. If the result is not empty (e.g. not $\perp$), then this indicates a potential overlap between the array indices. The constraint of the kind $(W.I = RW.I)$ corresponds to the property we are trying to refute, namely that the indices are equal. In addition, we add the global constraint that requires that task identifiers are distinct. The reason why we write that constraint as $(t_1 - t_2 \geq 1)$ as opposed to $(t_1 - t_2 > 0)$ is that the first form is precisely expressible in both Octagon and Polyhedra, while the second form is only expressible in Polyhedra. We assume that primitive variables from two different tasks have distinct names.

The definition of abstract conflicting accesses leads to a natural definition of *abstract pairwise conflicting program* based on Definition 4. Due to the soundness of our abstraction it follows that if we establish the program as abstract (pairwise) conflict free, then it is (pairwise) conflict free under the concrete semantics.

In the next section, we describe our implementation which is based on a sequential analysis of each task, computing the reachable abstract states of a task in the absence of interference from other tasks. We then (conservatively) check that tasks perform independent memory accesses. When tasks may be performing conflicting memory accesses, the sequential information computed may be invalid, and our analysis will not be able to establish determinism of the program. When tasks are only performing non-conflicting memory accesses, the information we compute sequentially for each task is stable, and we can use it to establish the determinism of the program.

5 Implementation

We implemented our analysis as a tool based on the Soot framework [36]. This allows us to potentially use many of the existing analyses already implemented in Soot, such as points-to analyses. The input to our tool is a Java program with annotations that denote the code of each task. In fact, as our core analysis is based purely on the Jimple intermediate representation produced by the Soot front end, as long as it knows what code each task executes, the analysis is applicable to standard concurrent Java programs.

The complete analysis works as follows:

Step 1: Apply Heap Abstraction. First, we apply the SPARK flow-insensitive pointer analysis on the whole program [20]. We note that flow-insensitive analysis is sound in the presence of concurrency, but as we will see later, the pointer analysis may be imprecise in most cases and hence we compute additional heap information (see the UniqueRef invariant later).

Step 2: Apply Numerical Abstraction. Second, for each task, we apply the appropriate (sequential) numerical analysis. Our numerical analysis uses the Java binding of the Apron library [15]. We initialized the environment of the analysis only with variables of integer type. As real variables cannot be used as array indices, they are ignored by the analysis. Currently, we do not handle casting from real to integer variables. However in our benchmarks we have not encountered such cast operations. The numerical constraints contain only variables of integer type.

Step 3: Compute MHP. Third, we compute the mhp predicate. In the annotated Java code that we consider, this is trivially computed as the annotations denote which tasks can execute in parallel and given that parallel tasks don't use any synchronization constructs internally, it implies that all statements in two parallel tasks can also execute in parallel. When analyzing standard Java programs which use synchronization primitives such as monitors, one can use an off-the-shelf MHP analysis (cf. [21,26]).

Step 4: Verify Conflict-Freedom. Finally, we check whether the program is conflict-free: for each pair of abstract states from two different tasks, we check whether that pair is conflict-free according to Definition 7. In our examples, it is often the case that the same code is executed by multiple tasks. Therefore, in our implementation, we simply check whether the abstract states of a single task are conflict-free with themselves. To perform the check, we make sure that local variables are appropriately renamed (the underlying abstract domain provides methods for this operation). Parameter variables that are common to all tasks that execute the same code maintain their name under renaming and are distinguished by special names. Note that our analysis verifies conflict-freedom between tasks in a pairwise manner, and does not make any assumption on the number of tasks in the system (thus also handling programs with an unbounded number of tasks).

5.1 Reference Arrays

Many parallel programs use reference arrays, usually multi-dimensional primitive arrays (e.g. `int A[][]`) or reference arrays of standard objects such as `java.lang.String`. In Jimple (and Java bytecodes), multi-dimensional arrays are represented via a hierarchy of one-dimensional arrays. Accesses to a k-dimensional array is comprised of k accesses to one-dimensional arrays. In many of our benchmarks, parallel tasks operate on disjoint portions of a reference array. However, often, given an array `int A[][]`, each parallel task accesses a different outer dimension, but accesses the internal array `int A[]` in the same way. For example, task 1 can write to `A[1][5]`, while task 2 can write to `A[2][5]`): the outer dimension (e.g. 2) is different, but the inner dimension (e.g. 5) is the same. The standard pointer analysis fails to establish that `A[1][5]` and `A[2][5]` are disjoint, and hence our analysis fails to prove determinism.

UniqueRef Global Invariant. However, in all of our benchmarks, the references inside reference arrays never alias. This is common among scientific computing benchmarks as they have a pre-initialization phase where they fill up the array, and thereafter, only the primitive values in the array are modified. To capture this, on startup, we perform a simple global analysis to establish that all writes of reference variables to cells in the reference array are only assigned to once with a fresh object, either a newly allocated object or a reference obtained as a result of a library call such as `java.lang.String.substring` that returns a fresh reference. While this simple treatment suffices for all of our benchmarks, general treatment of handling references inside objects may require more elaborate heap analysis.

Once we establish this invariant, we can either refine the pointer analysis information (to know that the inner dimensions of an array are distinct), or we can use the invariant directly in the analysis. In almost all of our benchmarks, we used this invariant directly.

Table 2. Experimental Results

Benchmark	Description	LOC	Vars	Domain	Iter	Time (s)	Widen	PA	Result
CRYPT	IDEA encryption	110	180	Polyhedra	439	54.8	No	No	✓
SOR	Successive over-relaxation	35	21	Polyhedra	72	0.41	Yes	No	✓
LUFACT	LU Factorization	32	22	Octagon	57	1.94	Yes	No	✓
SERIES	Fourier coefficient analysis	67	14	Octagon	22047	55.8	No	No	✓
MOLDYN1	Molecular dynamics simulation	85	18	Octagon	85	24.6	No	No	✓
MOLDYN2		137	42	Polyhedra	340	2.5	Yes	Yes	✓
MOLDYN3		31	8	Octagon	78	0.32	Yes	No	✓
MOLDYN4		50	10	Polyhedra	50	1.01	No	No	✓
MOLDYN5		37	18	Polyhedra	37	0.34	No	No	✓
SPARSE	Sparse matrix multiplication	29	17	Polyhedra	45	0.2	Yes	Yes	✗
RAYTRACER	3D Ray Tracer	-	-	-	-	-	-	-	✗
MONTECARLO	Monte Carlo simulation	-	-	-	-	-	-	-	✗

6 Evaluation

To evaluate our analysis, we selected the JGF benchmarks used by the HJ suite [1]. These benchmarks are modified versions of the Java JGF benchmarks [9]. As currently our numerical analysis is intra-procedural, we have slightly modified these benchmarks by inlining some of the function calls. The code for all benchmarks is available in [1].

Our analysis works on the Jimple intermediate representation, which is a three-address code representation for Java. Working at the Jimple level enables us to use standard analyses implemented in Soot, such as the Spark points-to analysis [20]. However, Jimple creates a large number of temporary variables, resulting in many more variables than the original Java source. This may lead to a larger number of numerical constraints compared to the ones arising when analyzing the program at the source level as in the case of the Interproc analyzer [16].

Analysis of some of our benchmarks required the use of widening. We used the `LoopFinder` API provided by Soot to identify loops and applied a basic widening strategy which only widens at the head of the loop and does so every k'th iteration, where k is a parameter to the analysis.

All of our experiments were conducted using a Java 1.6 runtime running on a 4-core Intel(R) Xeon(TM) CPU 3.80GHz processor with 5GB.

6.1 Results

Table 2 summarizes the results of our analysis. The columns indicate the benchmark name and description, lines of code for the analyzed program, the number of integer-valued variables used in the analysis, the numerical domain used, the number of iterations it took for the analysis to reach a fixed point, the combined time of the numerical

analysis and verification checking (pointer analysis time is not included even if used), whether the analysis needed widening to terminate, whether we used Spark pointer analysis (note that we almost always use the UniqueRef invariant as the programs make heavy use of multi-dimensional arrays), and the result of the analysis where ✓ denotes that it successfully proved determinism, and ✗ denotes that it failed to do so. As mentioned earlier, in our benchmarks, it is easy to pre-compute the mhp predicate and determine which tasks can run in parallel. That is, there is no need to perform numerical analysis on tasks that can never run in parallel with other tasks. Therefore, the lines of code in the table refer only to the relevant code that may run in parallel and is analyzed by the numerical analysis. The actual applications contain many more lines of code (in the range of thousands), as they need to pre-initialize the computation, but such initialization code never runs in parallel with other tasks.

Applying the analysis. For every benchmark, we first attempted to verify determinism with the simplest available configuration: e.g. Octagon domain without widening or pointer analysis. If the analysis did not terminate within 10 minutes, or failed to prove the program deterministic, then we tried adding widening and/or changing the domain to Polyhedra and/or performing pointer analysis. Usually, we did not find the need for using Spark. Instead, we almost always rely on the UniqueRef invariant.

For five of the benchmarks, the analysis managed to prove determinism, while it failed to do so for three benchmarks. Next, we elaborate on the details.

CRYPT involves reading and updating multiple shared one-dimensional arrays. This is a computationally intensive benchmark and its intermediate representation contains many variables. When we used the Octagon domain without widening, the analysis did not terminate and the size of the constraints kept growing. Even after applying our widening strategy (widening at the head of the loop) with various frequencies (e.g. the parameter *k* mentioned earlier), we still could not get the analysis to terminate. Only after applying very aggressive widening: in addition to every loop head, to widen at some points in the loop body, did we get the analysis to terminate. But even when it terminated, the analysis was unable to prove determinism. The key reason is that the program computes array indices for each task based on the task identifier via statements such as $ix_i = 8{*}tid_i$, where ix_i is the index variable and tid_i is the task identifier variable. Such constraints cannot be directly expressed in the Octagon domain. However, by using the Polyhedra domain, the analysis managed to terminate without widening. It managed successfully to capture the simple loop exit constraint $ix_i \geq k$ (even with the loop body performing complicated updates). It also managed to successfully preserve constraints such as $ix_1 = 8 * tid_1$. As a result, the computed constraints were precise enough to prove the program deterministic, which is the result that we report in the table.

In SOR, without widening, both Octagon and Polyhedra failed to terminate. With widening, Octagon failed to prove determinism due to the use of array index expressions such as $ix_i = 2 * tid_i - v$, where tid_i is the task identifier variable and v is a parameter variable. Constraints, such as $i_i = k * tid_i$, where $k > 1$ cannot be expressed in the Octagon domain and hence the analysis fails. Using Polyhedra with widening quickly succeed in proving determinism.

Without widening and with Octagon, the analysis did not terminate in LUFACT. However, with widening and Octagon, the analysis quickly reached a fixed point. The SERIES benchmark succeeds only with Octagon and required no widening but it took the longest to terminate.

MOLDYN contains a sequence of five blocks where only the tasks inside each block can run in parallel and tasks from different blocks cannot run in parallel. Interestingly, each block of tasks can be proved deterministic by using different configurations of domain, widening and pointer analysis. In Table 2, we show the result for each block as a separate row in the table. In the first block, tasks execute straight line code and determinism can be proved only with Octagon and no widening. In the second block, tasks contain loops and require Polyhedra, widening and pointer analysis. Without widening, both Octagon and Polyhedra do not terminate. With widening, Octagon terminates, but fails. The problem is that the array index variable ix_i is computed with the statement $ix_i = k * tid_i$, where k is a constant and $k > 1$. The Octagon domain cannot accurately represent abstract elements with such constraints. We used the pointer analysis to establish that references loaded from two different arrays are distinct, but we could have also computed that with the UniqueRef invariant. Tasks in the third block succeed with Octagon but also required widening. Tasks in the fourth and fifth blocks do not need widening (there are no loops), but require Polyhedra as they are using constraints such as $ix_i = k * tid_i$, where $k > 1$.

In SPARSE, the Polyhedra fails as the tasks use array indices obtained from other arrays, e.g. $A[B[i]]$, where the values of $B[i]$ are initialized on startup. The analysis required widening to terminate, but is unable to establish anything about $B[i]$ and hence cannot prove independence of two different array accesses $A[j]$ and $A[k]$, where j and k come from some $B[i]$.

In RAYTRACER, analysis fails as the program involves non-linear constraints and also uses atomic sections, which our analysis currently does not handle.

As mentioned, our analysis is intra-procedural. However, unlike the other benchmarks, MONTECARLO makes many nested calls and it would have been very error-prone to try and inline all of these nested calls. To handle such cases, in the future, we plan to extend our analysis to handle procedures.

6.2 Summary

In summary, in all cases where the analysis was successful in proving determinacy, it finished in under a minute. Different benchmarks could be proved with different combination of domain (Octagon or Polyhedra) and widening (to widen or not). In fact, the suite exercised all four combinations. In general, we did not find that we needed expensive pointer analysis, and it was sufficient to have the simple invariant that all arrays contain unique references, which was easily verifiable for our benchmarks (but in general may be a very hard problem). In cases where we failed, the program was using features that we do not currently handle such as: non-linear constraints, atomic sections, procedure calls or required maintaining scalar invariants over arrays (e.g. that integers inside an array are distinct). In the future, we plan to address these issues. This would also allow us to handle the full Java JGF benchmarks [9], where many benchmarks make use of such features.

7 Related Work

Recent papers by Burnim and Sen [5] and Sadowski et. al [33] focus on checking determinism dynamically. The first work focuses on user-defined notion of observationally equivalent states while the second paper checks for absence of conflicts. While both of these works are only able to dynamically *test* for determinism, our work focus on statically proving determinism.

There is a vast literature on dependence analysis for automatic parallelization (see e.g., [24, Sec. 9.8]). The focus of these analyses is on *efficiently* identifying independent loop iterations that can be performed in parallel. In contrast, our work focuses on verifying determinism of parallel programs. This usually involves dependence checking between tasks that may execute in parallel (and not necessarily in a loop). As we focus on verification, we can employ precise (and often expensive) numerical domains.

There is a large volume of work on dependence analysis for heap-manipulating programs (e.g., [14]), which at the end boils down to having a sufficiently precise heap abstraction (e.g., [29]). The current task-parallel applications we are dealing with mostly involve numerical computations over arrays. For such programs, simple heap abstractions were sufficient for establishing determinism. In the future, we plan to integrate more advanced heap abstractions into our analysis framework.

In [31,32], Rugina and Rinard present an analysis framework for computing symbolic bounds on pointer, array indices, and accesses memory regions. In order to support challenging features such as pointer arithmetic, their analysis framework requires an expensive flow-sensitive and context-sensitive pointer analysis [30] as a preceding phase. In our (simpler) setting, this is not required.

In [12], Ferrera presents a static analysis for establishing determinism of concurrent programs. The idea is to record for each variable what thread wrote it. This instrumented concrete semantics is then abstracted to an abstract semantics that records separately the value written to a variable by each thread. Determinism is established by comparing (abstract) states and showing that for each variable, its value is only determined by a single thread. The paper does not handle arrays, dynamic memory allocation, and assumes a bounded number of threads. Further, Ferrera's analysis is based on concurrent executions and therefore has to consider all possible interleavings. In contrast, using basic assume-guarantee reasoning, our analysis reduces the problem to a sequential analysis.

8 Conclusion

We present a static analysis for automatically verifying determinism of structured parallel programs. Our approach uses *sequential analysis* to establish that tasks that may execute in parallel only perform non-conflicting memory accesses. Our sequential analysis combines information about the heap with information about array indices to show that memory accesses are non-conflicting. We show that in realistic programs, establishing that accesses are non-conflicting requires powerful numerical domains such as Octagon and Polyhedra. We implemented our approach in a tool called DICE and applied it to verify determinism of several non-trivial benchmark programs. In the future, we plan to extend our analysis to handle general Java programs.

Acknowledgements. We thank the anonymous reviewers for their helpful comments that improved the paper, and Antoine Mine for helping us with using Apron.

References

1. Dojo: Ensuring determinism of concurrent systems, https://researcher.ibm.com/researcher/view_project.php?id=1337
2. Banerjee, U. K. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell (1988)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: PPoPP, pp. 207–216 (October 1995)
4. Bocchino, R., Adve, V., Adve, S., Snir, M.: Parallel programming must be deterministic by default. In: First USENIX Workship on Hot Topics in Parallelism (HOTPAR 2009) (2009)
5. Burnim, J., Sen, K.: Asserting and checking determinism for multithreaded programs. In: ESEC/FSE 2009, pp. 3–12. ACM, New York (2009)
6. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA, pp. 519–538 (October 2005)
7. Cousot, P., Halbwachs, N.: 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, Tucson, Arizona, pp. 84–97. ACM Press, New York (1978)
8. Devietti, J., Lucia, B., Ceze, L., Oskin, M.: Dmp: deterministic shared memory multiprocessing. In: ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 85–96. ACM Press, New York (2009)
9. Edinburgh Parallel Computing Centre. Java grande forum benchmark suite, http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html
10. Edwards, S.A., Tardieu, O.: Shim: a deterministic model for heterogeneous embedded systems. In: EMSOFT 2005: Proceedings of the 5th ACM International Conference on Embedded Software, pp. 264–272. ACM, New York (2005)
11. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in cilk programs. In: SPAA 1997: Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 1–11. ACM, New York (1997)
12. Ferrara, P.: Static analysis of the determinism of multithreaded programs. In: Proceedings of the Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008). I. C. Society, Los Alamitos (November 2008)
13. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 121–133. ACM, New York (2009)
14. Horwitz, S., Pfeiffer, P., Reps, T.: Dependence analysis for pointer variables. In: PLDI 1989: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language Design and Implementation, pp. 28–40. ACM, New York (1989)
15. Jeannet, B., Mine, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
16. Lalire, G., Argoud, M., Jeannet, B.: The interproc analyzer, http://pop-art.inrialpes.fr/interproc/interprocweb.cgi

17. Lamport, L.: The parallel execution of do loops. ACM Commun. 17(2), 83–93 (1974)
18. Lea, D.: A java fork/join framework. In: JAVA 2000: Proceedings of the ACM 2000 Conference on Java Grande, pp. 36–43. ACM, New York (2000)
19. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
20. Lhoták, O., Hendren, L.: Scaling java points-to analysis using spark. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
21. Li, L., Verbrugge, C.: A practical MHP information analysis for concurrent java programs. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 194–208. Springer, Heidelberg (2005)
22. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: PLDI 2009, pp. 134–143. ACM, New York (2009)
23. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. 19(1), 31–100 (2006)
24. Muchnick, S.S.: Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco (1997)
25. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 308–319. ACM, New York (2006)
26. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing MHP information for concurrent Java programs. In: Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 338–354 (September 1999)
27. O'Callahan, R., Choi, J.-D.: Hybrid dynamic data race detection. In: PPoPP 2003: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 167–178. ACM, New York (2003)
28. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. In: ASPLOS 2009, pp. 97–108. ACM, New York (2009)
29. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 348–362. Springer, Heidelberg (2009)
30. Rugina, R., Rinard, M.: Automatic parallelization of divide and conquer algorithms. In: PPoPP 1999: Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 72–83. ACM, New York (1999)
31. Rugina, R., Rinard, M.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: PLDI 2000: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 182–195. ACM, New York (2000)
32. Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. ACM Trans. Program. Lang. Syst. 27(2), 185–235 (2005)
33. Sadowski, C., Freund, S.N., Flanagan, C.: SingleTrack: A dynamic determinism checker for multithreaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 394–409. Springer, Heidelberg (2009)
34. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
35. Shacham, O., Sagiv, M., Schuster, A.: Scaling model checking of dataraces using dynamic information. In: PPoPP 2005, pp. 107–118. ACM Press, New York (2005)
36. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a java optimization framework. In: Proceedings of CASCON 1999, pp. 125–135 (1999)

Author Index

Aiken, Alex 236
Albert, Elvira 100
Alias, Christophe 117
Allen Emerson, E. 1
Amato, Gianluca 134
Appel, Andrew W. 151
Arenas, Puri 100

Bell, Christian J. 151
Blanco, Javier 201
Brauer, Jörg 167

Chaki, Sagar 287
Chapoutot, Alexandre 184
Cherini, Renato 201
Coogan, Kevin 218

Dalla Preda, Mila 218
Darte, Alain 117
Debray, Saumya 218
Dillig, Isil 236
Dillig, Thomas 236

Fähndrich, Manuel 2
Farzan, Azadeh 253
Feautrier, Paul 117

Gawlitza, Thomas Martin 271
Genaim, Samir 100
Giacobazzi, Roberto 218
Goldberg, Benjamin 6
Gonnord, Laure 117
Govindarajan, Ramaswamy 422
Gurfinkel, Arie 287

Harris, William R. 304
Heizmann, Matthias 22
Hofmann, Martin 340

Jensen, Simon Holm 320
Jones, Neil D. 22

Karbyshev, Aleksandr 340
Katoen, Joost-Pieter 390
Kincaid, Zachary 253
King, Andy 167

Lal, Akash 304
Lesens, David 51

Malkis, Alexander 356
Matringe, Nadir 373
McCloskey, Bill 71
McIver, Annabelle K. 390
Meinicke, Larissa A. 390
Might, Matthew 407
Møller, Anders 320
Morgan, Carroll C. 390
Moura, Arnaldo Vieira 373
Mycroft, Alan 439

Nasre, Rupesh 422
Nori, Aditya V. 304

Parton, Maurizio 134
Podelski, Andreas 22, 356
Puebla, German 100

Rajamani, Sriram K. 304
Raman, Raghavan 455
Ramírez Deantes, Diana Vanessa 100
Rearte, Lucas 201
Rebiha, Rachid 373
Reps, Thomas 71
Rybalchenko, Andrey 356

Sagiv, Mooly 71
Sarkar, Vivek 455
Schrijvers, Tom 439
Scozzari, Francesca 134
Seidl, Helmut 271, 340

Thiemann, Peter 320
Townsend, Gregg M. 218

Vechev, Martin 455

Walker, David 151

Yahav, Eran 455