Hongseok Yang (Ed.)

Programming Languages and Systems

LNCS 7078

9th Asian Symposium, APLAS 2011
Kenting, Taiwan, December 2011
Proceedings

Lecture Notes in Computer Science 7078

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

Hongseok Yang (Ed.)

Programming Languages and Systems

9th Asian Symposium, APLAS 2011
Kenting, Taiwan, December 5-7, 2011
Proceedings

Volume Editor

Hongseok Yang
University of Oxford
Department of Computer Science
Parks Road
Oxford OX1 3QD, UK
E-mail: hongseok.yang@cs.ox.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-25317-1 e-ISBN 978-3-642-25318-8
DOI 10.1007/978-3-642-25318-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011941029

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at APLAS 2011, the 9th Asian Symposium on Programming Languages and Systems held during December 5–7, 2011 in Kenting, Taiwan. The symposium was sponsored by the Asian Association for Foundation of Software (AAFS), Academia Sinica (Taiwan), and National Taiwan University. We are grateful for the administrative support from the Institute of Information Science, Academia Sinica (Taiwan), and the Department of Information Management and the Yen Tjing Ling Industrial Research Institute at National Taiwan University.

APLAS is a premier forum for the discussion of a wide range of topics related to programming languages and systems. Although it is based in Asia, APLAS has been an international forum that serves the worldwide programming language community. The past APLAS symposiums were successfully held in Shanghai (2010), Seoul (2009), Bangalore (2008), Singapore (2007), Sydney (2006), Tsukuba (2005), Taipei (2004) and Beijing (2003) after three informal workshops. Proceedings of the past symposiums were published in Springer Verlag's LNCS series as volumes 6461, 5904, 5356, 4807, 4279, 3780, 3302, and 2895 respectively.

Following the initiative from the previous year, APLAS 2011 solicited submissions in two categories, *regular research papers* and *system and tool presentations*. There were 64 submissions from 22 countries (62 regular research papers and 2 system and tool presentations). Each submission was reviewed by at least 2, and on average 3.2, Program Committee members with the help of external reviewers. The Program Committee meeting was conducted electronically over a period of two weeks in August 2011. The Program Committee decided to accept 22 regular research papers (35%) and 1 system and tool presentation (50%). Among the 22 accepted papers, there was one whose initial verdict was conditional acceptance. To have their paper accepted, the authors were requested to address specific concerns raised by the Program Committee. The revised version of the paper was checked by the Program Committee, before it was finally accepted. I would like to thank all the Program Committee members for their hard work in reviewing papers, participating in online discussions, volunteering to shepherd submissions, and sometimes finding and fixing technical errors in submissions. I also want to thank all the external reviewers for their invaluable contributions.

In addition to contributed papers, this volume contains the full paper or the extended abstracts of of four distinguished invited speakers: Nikolaj Bjørner (Microsoft Research Redmond), Ranjit Jhala (University of California, San Diego), Peter O'Hearn (Queen Mary University of London) and Sriram Rajamani (Microsoft Research India). I would like to thank all of these speakers for accepting our invitation and contributing papers or abstracts.

The General Chair, Tyng-Ruey Chuang, helped me so much from the very beginning while I prepared the technical program of APLAS 2011. I am truly grateful for his support and guidance, and also for making our symposium in Kenting possible and enjoyable. I would like to thank Shin-Cheng Mu for local arrangements, and Yih-Kuen Tsay for serving as the Finance Chair. Noam Rinetzky was an excellent Publicity Chair for APLAS 2011, who had several creative ideas for increasing the awareness of the conference, and Mike Dodds worked very hard to organize the poster session. I greatly appreciate their help and efforts. EasyChair made the handling of submissions and the productions of the proceedings extremely smooth and efficient. Finally, I would like to thank the members of the AAFS Executive Committee for their advice. In particular, Kazunori Ueda, the Program Chair of APLAS 2010, gave me invaluable tips and suggestions, without which I would not have been able to prepare the technical program and the proceedings of APLAS 2011.

September 2011 Hongseok Yang

Organization

Program Committee

Lars Birkedal	IT University of Copenhagen, Denmark
James Brotherston	Imperial College London, UK
Kung Chen	National Chengchi University, Taiwan
Wenguang Chen	Tsinghua University, China
Wei-Ngan Chin	National University of Singapore, Singapore
Javier Esparza	Technische Universität München, Germany
Xinyu Feng	University of Science and Technology of China, China
Jerome Feret	INRIA, France
Matthew Fluet	Rochester Institute of Technology, USA
Rajiv Gupta	Univeristy of California Riverside, USA
Masahito Hasegawa	Kyoto University, Japan
Radha Jagadeesan	DePaul University, USA
Naoki Kobayashi	Tohoku University, Japan
Daniel Kroning	University of Oxford, UK
Rupak Majumdar	Max Planck Institute for Software Systems, Germany
Andrzej Murawski	University of Leicester, UK
Paulo Oliva	Queen Mary University of London, UK
Doron Peled	Bar Ilan University, Israel
Sukyoung Ryu	KAIST, Korea
Sriram Sankaranarayanan	University of Colorado, USA
Armando Solar-Lezama	MIT, USA
Sam Staton	University of Cambridge, UK
Viktor Vafeiadis	Max Planck Institute for Software Systems, Germany
Kapil Vaswani	Microsoft Research, India
Martin Vechev	ETH Zurich, Switzerland
Peng Wu	IBM T.J. Watson Research Center, USA
Hongseok Yang	University of Oxford, UK
Pen-Chung Yew	Academia Sinica, Taiwan

Additional Reviewers

Asada, Kazuyuki	Berger, Ulrich	Charan K, Sai
Bengtson, Jesper	Bocchino, Robert	Charlton, Nathaniel
Berdine, Josh	Camporesi, Ferdinanda	Chen, Yu-Fang
Berger, Martin	Chang, Bor-Yuh Evan	Chu, Duc Hiep

Costea, Andreea
Daniele, Varacca
Debois, Soren
Dimoulas, Christos
Fu, Ming
Gaiser, Andreas
Gherghina, Cristian
Guha, Arjun
Gupta, Ashutosh
Haack, Christian
Haller, Leopold
Hancock, Peter
Hardekopf, Ben
Hoshino, Naohiko
Jiang, Xinyu
Katsumata, Shin-Ya
Kern, Christian
Khoo, Siau-Cheng
Klin, Bartek
Krishnaswami, Neelakantan
Landsberg, David
Lee, Jenq-Kuen
Liang, Hongjin
Lin, Changhui
Longley, John
Maietti, Maria Emilia
Mehnert, Hannes
Mogelberg, Rasmus
Mu, Shin-Cheng
Nakata, Keiko
Nanevski, Aleks
Nordlander, Johan
Petter, Michael
Piérard, Adrien
Rauchwerger, Lawrence
Rival, Xavier
Romanel, Alessandro
Rossberg, Andreas
Sanchez, Alejandro
Sangiorgi, Davide
Schwinghammer, Jan
Schwoon, Stefan
Seghir, Mohamed Nassim
Sharma, Asankhaya
Sobocinski, Pawel
Strassburger, Lutz
Streicher, Thomas
Suenaga, Kohei
Tan, Li
Tautschnig, Michael
Wang, Meng
Wang, Yan
Weng, Shu-Chun
Xu, Na
Xue, Jingling
Yang, Jean

Table of Contents

Invited Talks

Program Analysis and Machine Learning: A Win-Win Deal 1
Aditya V. Nori and Sriram K. Rajamani

Software Verification with Liquid Types (Abstract) 3
Ranjit Jhala

Engineering Theories with Z3 4
Nikolaj Bjørner

Algebra, Logic, Locality, Concurrency 17
Peter W. O'Hearn

Session 1: Program Analysis

Modular Abstractions of Reactive Nodes Using Disjunctive Invariants 19
David Monniaux and Martin Bodin

Template-Based Unbounded Time Verification of Affine Hybrid Automata 34
Thao Dang and Thomas Martin Gawlitza

Access-Based Localization with Bypassing 50
Hakjoo Oh and Kwangkeun Yi

A Deductive Database with Datalog and SQL Query Languages 66
Fernando Sáenz-Pérez, Rafael Caballero, and Yolanda García-Ruiz

Session 2: Functional Programming

Constructing List Homomorphisms from Proofs 74
Yun-Yan Chi and Shin-Cheng Mu

Extending Hindley-Milner Type Inference with Coercive Structural Subtyping 89
Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow

Polymorphic Multi-stage Language with Control Effects 105
Yuichiro Kokaji and Yukiyoshi Kameyama

Session 3: Compiler

Compiler Backend Generation for Application Specific Instruction Set Processors 121
Zhen Cao, Yuan Dong, and Shengyuan Wang

A Non-iterative Data-Flow Algorithm for Computing Liveness Sets in Strict SSA Programs 137
Benoît Boissinot, Florian Brandner, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello

SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA 155
Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew

Session 4: Concurrency 1

On the Strength of Owicki-Gries for Resources 172
Alexander Malkis and Laurent Mauborgne

Solving Recursion-Free Horn Clauses over LI+UIF 188
Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko

Macro Tree Transformations of Linear Size Increase Achieve Cost-Optimal Parallelism 204
Akimasa Morihata

Decentralized Delimited Release 220
Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld

Session 5: Concurrency 2

Cost Analysis of Concurrent OO Programs 238
Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and German Puebla

Static Object Race Detection 255
Ana Milanova and Wei Huang

Soundness of Data Flow Analyses for Weak Memory Models 272
Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig

Session 6: Semantics

Towards a General Theory of Barbs, Contexts and Labels 289
Filippo Bonchi, Fabio Gadducci, and Giacoma Valentina Monreale

Computation-by-Interaction with Effects 305
Ulrich Schöpp

Session 7: Certification and Logic

Towards a Certified Petri Net Model-Checker 322
Lukasz Fronc and Franck Pommereau

Elementary Linear Logic Revisited for Polynomial Time and an Exponential Time Hierarchy 337
Patrick Baillot

A Proof Pearl with the Fan Theorem and Bar Induction: Walking through Infinite Trees with Mixed Induction and Coinduction 353
Keiko Nakata, Tarmo Uustalu, and Marc Bezem

A Semantics for Context-Sensitive Reduction Semantics 369
Casey Klein, Jay McCarthy, Steven Jaconette, and Robert Bruce Findler

Author Index 385

Program Analysis and Machine Learning: A Win-Win Deal

Aditya V. Nori and Sriram K. Rajamani

Microsoft Research India
{adityan,sriram}@microsoft.com

We give an account of our experiences working at the intersection of two fields: program analysis and machine learning. In particular, we show that machine learning can be used to infer annotations for program analysis tools, and that program analysis techniques can be used to improve the efficiency of machine learning tools.

Every program analysis tool needs annotations. Type systems need users to specify types. Program verification tools need users to specify preconditions, postconditions and invariants in some form. Information flow analyzers require users to specify sources and sinks for taint, and sanitizers, which cleanse taint. We show how such annotations can be derived from high level intuitions using Bayesian inference. In this approach, annotations are thought of as random variables, and intuitions of the programmer are stated as probabilistic constraints over these random variables. The Bayesian framework models and tolerates uncertainty in programmer intuitions, and Bayesian inference is used to infer most likely annotations, given the program structure and programmer intuitions. We give specific examples of such annotation inference for information flow [5] and ownership types [1]. We also describe a generic scheme to infer annotations for any safety property.

Machine learning algorithms perform statistical inference by analyzing voluminous data. Program analysis techniques can be used to greatly optimize these algorithms. In particular, statistical inference tools [3,6] perform inference from data and first-order logic specifications. We show how Counterexample Guided Abstraction Refinement (CEGAR) techniques, commonly used in verification tools and theorem provers can be used to lazily instantiate axioms and improve the efficiency of inference [2]. This approach also enables users of these tools to express their models with rich theories such as linear arithmetic and uninterpreted functions. There is a recent trend in the machine learning community to specify machine learning models as programs [4]. Inspired by this view of models as programs, we show how program analysis techniques such as backward analysis and weakest preconditions can be used to improve the efficiency of algorithms for learning tasks such as the computation of posterior probabilities given some observed data.

In summary, we believe that these cross fertilization of ideas from program analysis and machine learning have the potential to improve both fields, resulting in a mutual win-win deal. We speculate on further opportunities for mutually beneficial exchange of ideas between the two fields.

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 1–2, 2011.

References

1. Beckman, N., Nori, A.V.: Probabilistic, modular and scalable inference of typestate specifications. In: Programming Languages Design and Implementation (PLDI), pp. 211–221 (2011)
2. Chaganty, A., Lal, A., Nori, A.V., Rajamani, S.K.: Relational learning modulo axioms. Technical report, Microsoft Research (2011)
3. Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Domingos, P.: The Alchemy system for statistical relational AI. Technical report, University of Washington, Seattle (2007), http://alchemy.cs.washington.edu
4. Koller, D., McAllester, D.A., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: Fifteenth National Conference on Artificial Intelligence (AAAI), pp. 740–747 (1997)
5. Livshits, V.B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: Specification inference for explicit information flow problems. In: Programming Languages Design and Implementation (PLDI), pp. 75–86 (2009)
6. Niu, F., Re, C., Doan, A., Shavlik, J.: Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. In: International Conference on Very Large Data Bases, VLDB (2011)

Software Verification with Liquid Types*

Ranjit Jhala

University of California, San Diego
jhala@cs.ucsd.edu

Abstract. Traditional software verification algorithms work by using a combination of Floyd-Hoare Logics, Model Checking and Abstract Interpretation, to check and infer suitable program invariants. However, these techniques are problematic in the presence of complex but ubiquitous constructs like generic data structures, first-class functions. We observe that modern type systems are capable of the kind of analysis needed to analyze the above constructs, and we use this observation to develop Liquid Types, a new static verification technique which combines the complementary strengths of Floyd-Hoare logics, Model Checking, and Types. As a result, we demonstrate how liquid types can be used to statically verify properties ranging from memory safety to data structure correctness, in higher-order languages like ML. This presentation is based on joint work with Patrick Rondon and Ming Kawaguchi.

* This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and gifts from Microsoft Research.

H. Yang (Ed.): APLAS 2011, LNCS 7078, p. 3, 2011.

Engineering Theories with Z3

Nikolaj Bjørner

Microsoft Research
nbjorner@microsoft.com

Abstract. Modern Satisfiability Modulo Theories (SMT) solvers are fundamental to many program analysis, verification, design and testing tools. They are a good fit for the domain of software and hardware engineering because they support many domains that are commonly used by the tools. The meaning of domains are captured by theories that can be axiomatized or supported by efficient *theory solvers*. Nevertheless, not all domains are handled by all solvers and many domains and theories will never be native to any solver. We here explore different theories that extend Microsoft Research's SMT solver Z3's basic support. Some can be directly encoded or axiomatized, others make use of user theory plug-ins. Plug-ins are a powerful way for tools to supply their custom domains.

1 Introduction

This paper surveys a selection of theories that have appeared in applications of Z3 [7] and also in recent literature on automated deduction. In each case we show how the theories can be supported using either existing built-in theories in Z3, or by adding a custom decision procedure, or calling Z3 as a black box and adding axioms between each call. The theme is not new. On the contrary, it is very central to research on either encoding (reducing) theories into a simpler basis or developing special solvers for theories. Propositional logic is the most basic such basis e.g., [13]. In the context of SMT (Satisfiability Modulo Theories), the basis is much richer. It comes with built-in support for the theory of equality, uninterpreted functions, arithmetic, arrays, bit-vectors, and even first-order quantification. The problem space is rich, and new applications that require new solutions keep appearing. We don't offer a silver bullet solution, but the "exercise" of examining different applications may give ideas how to tackle new domains.

Z3 contains an interface for plugging in custom theory solvers. We exemplify this interface on two theories: MaxSMT (Section 3) and partial orders (Section 4). This interface is powerful, but also requires thoughtful interfacing. To date it has been used in a few projects that we are aware of [17,2,15]. Some of our own work can also be seen as an instance of a theory solver. The quantifier-elimination procedures for linear arithmetic and algebraic data-types available in Z3 acts as a special decision procedure [3]. The OpenSMT solver also supports

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 4–16, 2011.

an interface for pluggable theories [5]. We feel that the potential is much bigger an we conclude with some speculation where the pluggable interface could be used elsewhere.

Z3 also allows interfacing theories in simpler ways. The simplest is by encoding and Section 5 discuss a simple theory with two encodings. Something between encoding and a user theory, is by calling Z3 repeatedly. Whenever Z3 returns a satisfiable state, then add new axioms that are not satisfied by the current candidate model for the existing formulas. Section 6 discusses how HOL can be solved using this method.

The case studies discussed in this paper are available as F# code samples.

2 SMT, DPLL(T), and Z3

2.1 SMT

We will not survey SMT here, but refer the reader to [8] for an introduction.

2.2 DPLL(T)

Modern SMT solvers are mostly based on the DPLL(T) architecture. In this context, an efficient propositional SAT solver is used to produce a truth assignment to the atomic sub-formulas of the current goal. Let us use M to refer to a partial assignment. It can be represented as a stack of literals $\ell_1, \ell_2, \ldots, \ell_n$. The partial assignment is updated by adding new literals to the stack to indicate their values, and by shrinking the stack. We use F for the current goal. The DPLL(T) architecture uses two main methods for interacting with a theory solver.

T-Propagate. Given a state $M \parallel F$, such that ℓ or $\neg\ell$ occurs in F, ℓ is unassigned in M, $\overline{C} \subseteq M$ (the negation of the literals in C are already assigned in M), and $T \vdash C \vee \ell$, then ℓ must be *true* under the current assignment M. It is then sound to propagate ℓ.

External theory solvers in Z3 can force this propagation by asserting the clause $(C \vee \ell)$. Then ℓ gets assigned by propositional propagation. However, the asserted clause has no value if ℓ does not participate in a conflict. So the default behavior in Z3 is to garbage collect the asserted clause on backtracking.

T-Conflict. Given a state $M \parallel F$ such that $\overline{C} \subseteq M$, $T \models C$. That is, there is a subset $\overline{C}$ of the literals in M that are inconsistent from the point of view of T, or dually the clause C is T-valid, then assert the valid clause C. The new clause is in conflict with the current assignment M, because all literals in C are *false* under M. The propositional engine detects the resulting conflict and causes backtracking.

The clause C may also be useless beyond serving the role of signaling the conflict. It is therefore also by default garbage collected during Z3's backtracking.

2.3 Z3's Theory Solver API

Z3 exposes a programmatic API for interfacing with the theory solver. It includes hooks for user theories to add callbacks so that they can implement the effect of T-Propagate and T-Conflict. As we saw, the effect of the rules is communicated by asserting a new clause to the current state. The corresponding method is called AssertTheoryAxiom over the .NET API. On the other hand, the state changes to the partial model M are exposed to the user solver using callbacks. When a literal is added to M (is assigned to either *true* or *false*), then a callback (called NewAssignment(*atom*,*truth_value*)) is invoked with the underlying atomic formula and the truth value it is assigned to. There are other specialized callbacks when new equalities and dis-equalities are discovered. Equalities and dis-equalities don't have to correspond to existing atoms. Finally, a callback (called FinalCheck in .NET) is invoked when the current assignment fully satisfies the current formula F from the point of view of the built-in theories in Z3. The user theory solver can inspect its own state and compare the assignment it learned from NewAssignment to determine if the resulting assignment is satisfiable. When the solver performs a new case split or backtracks through states it calls into the theory solver with callbacks Push and Pop. Any side-effects made in a user-theory inside the scope of a Push need to be undone when receiving a matching call to Pop. For example, if a user-theory performs an update $c \leftarrow c + w_i$, where w_i is a constant, then it can undo the effect of the operation by executing $c \leftarrow c - w_i$ during a Pop.

3 Weighted MaxSMT

Weighted MaxSMT is the following problem. Given a set of numeric *weights* $w_1, \ldots, w_n$ and formulas $F_0, F_1, \ldots, F_n$, find the subset $I \subseteq \{1, \ldots, n\}$ such that

1. $F_0 \wedge \bigwedge_{i \notin I} F_i$ is satisfiable.
2. The *cost*: $\Sigma_{i \in I} w_i$ is minimized.

In other words, the weight w_i encodes the penalty for a formula F_i to not be included in a satisfying assignment. The paper [14] develops a theory solver for weighted MaxSMT. An important point is that the theory evolves as search progresses: once a satisfiable state is reached with a given cost c, then assignments that meet or exceed c are useless.

According to [14], weighted MaxSMT can be encoded in Z3 in the following way: Initially we assert F_0 and $F_i \vee p_i$ for each i, where p_i is a fresh propositional variable. We also maintain a cost c that is initialized to 0, and a *min_cost* that is set to *nil*. Then, repeat the following steps until the asserted formulas are unsatisfiable:

1. When some p_i is assigned to *true*, then update $c \leftarrow c + w_i$.
2. If $nil \neq min\_cost \leq c$, then block the current state by calling AssertTheoryAxiom($\bigvee\{\neg p_i \mid p_i$ is assigned to *true*$\}$).
3. When receiving the FinalCheck callback, it must be the case that $c < min\_cost$ or min_cost is *nil*. So it is safe to set $min\_cost \leftarrow c$. To block this current cost call AssertTheoryAxiom($\bigvee\{\neg p_i \mid p_i$ is assigned to *true*$\}$).

It was tempting to make fuller use of theory support in Z3 for MaxSMT. We also tried an encoding that used extra variables $v_1, \ldots, v_n$ and axioms $p_i \implies v_i = w_i$, $\neg p_i \implies v_i = 0$, $0 \leq v_i$ for each $i = 1, \ldots, n$ and $\Sigma_i v_i < B$. The idea would be to add assertions $B \leq c$ every time a new satisfying state with cost c is encountered. This encoding was counter-productive on the benchmarks used in [14], it taxes Z3's arithmetic solver in contrast to relatively cheap propagation using the blocking propositional clauses.

It can be highly domain dependent whether a particular solution applies. An example of constraints where the proposed MaxSMT solver performs poorly comes from constraints used to tune parameters for the Vampire [12] theorem prover. There, a formula $F_0[v_1, \ldots, v_n]$ is asserted and the goal is to minimize $\Sigma_i v_i$. The domain of the variables is bounded and the so-far best encoding appears to be to use bit-vectors for the variables. We can convert the problem to MaxSMT by adding the following soft clauses: $v_i[k] = 0$ with weight 2^k for each variable v_i of bit-width N and $0 \leq k < N$. Nevertheless, we found that this encoding is inferior to the best technique known so far: a binary search over constraints of the form $\Sigma_i v_i > c$, where c is a candidate lower bound.

4 Theories for Partial Orders and Class Inheritance

A partial order is a binary relation that is reflexive, anti-symmetric, and transitive. In other words, $\preceq$ is a partial order if for every x, y, z:

$$x \preceq x, \quad x \preceq y \wedge y \preceq x \implies x = y, \quad x \preceq y \wedge y \preceq z \implies x \preceq z$$

When there are no other non-ground properties of $\preceq$, it is relatively straightforward to support the theory using axioms that get instantiated fully during search. Unfortunately, the theory is *expensive*. When n is the number of terms in the goal that occur in either side of $\preceq$, the axiom for transitivity causes up to $O(n^3)$ clauses and generates up to $O(n^2)$ instantiations of $\preceq$. The (quantifier-free) theory of partial orders can be solved using graph search procedures. Let us illustrate two theory solvers in the context of partial orders.

4.1 A Basic Solver for Partial Orders

We present a basic decision procedure for the theory of partial orders. It maintains a directed graph $\mathcal{D}$ and a set $\mathcal{N}$ of pairs of terms. They are both initially

empty. Assert F, the original formula to satisfy and check for satisfiability with the following theory solver actions:

1. When $t \preceq t'$ is asserted to *true*, then add the edge $t \rightarrow t'$ to $\mathcal{D}$.
2. When $t = t'$ is asserted, then add edges $t \rightarrow t' \rightarrow t$ to $\mathcal{D}$.
3. If $\mathcal{D}$ contains a cycle with edges $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow ..t_n \rightarrow t_1$, for terms that are not yet asserted equal, then call

$$\mathsf{AssertTheoryAxiom}(t_1 \preceq t_2 \preceq t_3 \ldots t_n \preceq t_1 \rightarrow \bigwedge_{i=1}^{n-1} t_i = t_{i+1})$$

4. When $t \preceq t'$ is asserted to *false*, then add the pair (t, t') to $\mathcal{N}$.
5. If for some pair (t, t') in $\mathcal{N}$ there is a path in $\mathcal{D}$ from t to t' (the path can be empty and $t = t'$), then assert

$$\mathsf{AssertTheoryAxiom}(t \preceq t_1 \preceq t_2 \ldots t_n \preceq t' \rightarrow t \preceq t')$$

Correctness of the algorithm is straight-forward: The graph $\mathcal{D}$ is a model: every term in strongly connected components are forced equal, and every constraint $t \not\preceq t'$ is checked. It is critical that the algorithm has access to the current equalities between terms and it takes part of equality propagation as well.

A basic implementation of the corresponding solver is to defer all processing to FinalCheck. Tarjan's ubiquitous linear time algorithm for finding strongly connected components in a graph will identify implied equalities, and each pair in $\mathcal{N}$ can be checked in time $|\mathcal{D}|$.

4.2 Sub-typing Closure

The object inheritance hierarchy of classes in object oriented programs forms a partial order. A special class of partial order constraints are relevant in this context, and [16] develops a specialized decision procedure. In the context of object inheritance we can assume there is a fixed set of constants $cls_1, \ldots, cls_n$ that are (1) all distinct and (2) covers the universe of types that are used in the query. The type hierarchy among $cls_1, \ldots, cls_n$ is fixed once and the queries are Boolean formulas over atoms of the form $x = cls_i$ and $x \preceq cls_i$, where x is a variable (it is equal to one of $cls_1, \ldots, cls_n$, but the concrete value is not known yet.

The theory can be handled using a specialized solver that tracks satisfiability of assignments to the atoms: For each variable x, initialize the set of candidates $cand(x)$ to $\{cls_1, \ldots, cls_n\}$, and dependencies $dep(x)$ to $\emptyset$.

1. The state is updated upon asserting a literal ℓ as follows:
   (a) $x = cls_i$: set $cand(x) \leftarrow cand(x) \cap \{cls_i\}$.
   (b) $x \neq cls_i$: set $cand(x) \leftarrow cand(x) \setminus \{cls_i\}$.
   (c) $x \preceq cls_i$: intersect $cand(x)$ with the descendants of cls_i.
   (d) $x \not\preceq cls_i$: subtract the descendants for cls_i from $cand(x)$.

2. The asserted literal ℓ is also added to $dep(x)$.
3. The state is unsatisfiable if $cand(x) = \emptyset$. To block it, call:

$$\mathsf{AssertTheoryAxiom}(\bigvee\{\neg\ell \mid \ell \in dep(x)\})$$

The interesting problem is implementing the updates to $cand(x)$ efficiently. The assumption that the type hierarchy is fixed can be exploited. The Type Slicing [10] structure was developed in the context of fast dispatch tables for object oriented programs, and it was used in [16] for making the updates to $cand(x)$ efficient. The data-structure represents a partial order (directed acyclic graph) using a set of colored nodes that are ordered. The data-structure satisfies the following condition: For every node n and color c, the set of descendants of n of color c are contiguous with respect to the ordering. The contiguity requirement allows to represent descendants using the first and last element only of the interval. We will not review this data-structure and the methods for building it here, but note that the sketched solver integration with Z3 allows writing only the theory solver, while efficient handling of Boolean case splitting comes for free.

Remark 1. When detecting a conflict we suggested to include the negation of all literals from $dep(x)$ in the asserted theory axiom. The resulting axiom may have redundancies. For example if we assert $x = \mathsf{string}$ followed by $x \preceq \mathsf{System.Object}$ followed by $x = \mathsf{bool}$, we obtain a conflict by just producing the clause $x \neq \mathsf{string} \vee x \neq \mathsf{bool}$. The constraint $x \preceq \mathsf{System.Object}$ is redundant. A simple method is to minimize the conflicting dependencies for x by temporarily removing each literal from $dep(x)$ and check if there is still a conflict. Generating minimal conflicts is important for efficient search.

5 A Theory of Object Graphs

There are many cases where a new theory can already be encoded using existing built-in theories. There is then no need for special purpose procedures. Still there may not be a unique way to encode these theories. We here give an example of this situation.

The theory of object graphs uses elements from the theory of algebraic and co-algebraic data-types, yet it is not possible to directly use one or the other. The theory is also non-extensional. The theory of object graphs occurs naturally in the context of Pex [11]. Pex is a state-of-the-art tool for unit-test case generation. It applies to typed .NET code. Let us here consider the following program fragment:

```
class O {

   public readonly D d;
   public readonly O left;
   public O right;

   public O(D data,
            O left,
            O right) {
     this.data = data;
     this.left = left;
     this.right = right;
   }
}
```

```
void f(O n0) {

  Assert (n0 == null ||
          n0.left != n0);

  O n1 = new O(1,null,null);
  O n2 = new O(2,n1,null);
  O n3 = new O(2,n1,null);

  Assert(n2 != n3);

  n1.right = n2;
  n2.right = n1;
...
```

Program 5.1:

Objects of type O are created using a constructor that we also call O. Each allocation creates a different object (the default equality method is reference equality), so in the program `n2` is different from `n3`. We can use a heap, here called H, to track the state of objects. So access and updates to objects is done through the heap. The signature that is relevant for O is:

sorts:	O,
constructors:	$null : O, O : H \times D \times O \times O \to H \times O$,
accessors:	$data : H \times O \to D, left : H \times O \to O, right : H \times O \to O$,
modifiers:	$update\_right : H \times O \times O \to H$

The sort is O and there is a distinguished constant *null*. There are three accessors, the *data* accessor retrieves a data field from objects of type O, and *left* and *right* access left and right children. The read-only declared attributes of O cannot be updated, so there is only a single modifier for the *right* attribute.

The theory of O is characterized as follows:

$$
\begin{aligned}
&(h', o) = O(h, d, l, r) \implies o \neq null \\
&(h', o) = O(h, d, l, r) \implies data(h', o) = d \\
&(h', o) = O(h, d, l, r) \implies left(h', o) = l \\
&(h', o) = O(h, d, l, r) \implies right(h', o) = r \\
&left(null) = right(null) = null \\
&h' = update\_right(h, o, r) \wedge o \neq null \implies right(h', o) = r \\
&h' = update\_right(h, o, r) \wedge o' \neq o \implies right(h', o') = right(h, o') \\
&h' = update\_right(h, o, r) \implies left(h', o') = left(h, o') \\
&h' = update\_right(h, o, r) \implies data(h', o') = data(h, o')
\end{aligned}
$$

The read-only field constrains what objects are possible in a valid heap state. In particular all formulas of the form

$$o \neq null \implies left(h_1, left(h_2, left(\ldots left(h_n, o)))) \neq o \tag{1}$$

are valid. The restriction is similar to the occurs check (well-foundedness) of recursive data-types. On the other hand, the attributes following paths using *right* need not be well-founded.

The question we will now address is: How can we equip a decision procedure for reasoning about ground formulas over the theory of O?

5.1 An Encoding Using Arrays

A direct encoding of objects is to use one array per field. To enforce well-foundedness of left-access (see (1)) one can use a time-stamp. We use $O \Rightarrow D$ for the sort of arrays that map O to D, and encode the sort O as the set N of natural numbers. The sort H is a tuple with one array for *data*, other arrays for *left* and *right*, and finally a clock that we will increment when allocating new objects.

$$\begin{aligned} O &= N \\ H &= \langle data : O \Rightarrow D, left : O \Rightarrow O, right : O \Rightarrow O, clock : N \rangle \end{aligned}$$

The constant *null* is set to 0 and object allocation modifies the arrays maintained in H. The initial heap h_0 uses the value 0 for *clock*, such that allocated objects are different from *null*.

$$\begin{aligned} null &= 0 \\ left_0 &= store(left_0, null, null) \\ right_0 &= store(right_0, null, null) \\ h_0 &= \langle data, left_0, right_0, 0 \rangle \\ O(\langle data, left, right, clock \rangle, d, l, r) &= \begin{pmatrix} let\ o = clock + 1 \\ (\langle store(data, o, d), store(left, o, l), \\ store(right, o, r), clock + 1 \rangle, o) \end{pmatrix} \\ data(\langle data, left, right, clock \rangle, o) &= select(data, o) \\ left(\langle data, left, right, clock \rangle, o) &= select(left, o) \\ right(\langle data, left, right, clock \rangle, o) &= select(right, o) \\ update\_right(\langle data, left, right, clock \rangle, o, r) &= \langle data, left, store(right, o, r), clock \rangle \end{aligned}$$

To enforce well-foundedness in models produced by Z3 it suffices to enforce that the time-stamp (here it is the same as the natural number used to identify objects) on non-null objects is smaller on their left children. It suffices to assert an axiom that gets instantiated for every use of $left(h, o)$[1]

$$\forall h : H, o : O \;.\; o \neq null \implies 0 \leq left(h, o) < o \;.$$

[1] The mechanism for achieving this in Z3 is to annotate quantified formulas using this term as a *pattern*.

5.2 An Encoding Using Recursive Data-Types and Arrays

Another option is to encode the read-only fields using the theory of algebraic data-types. We use a unique identifier field *id* to make sure allocated objects are distinct.

$$O = null \mid O(id : N, data : D, left : O)$$
$$H = \langle right : O \Rightarrow O, clock : N \rangle$$

$$\begin{aligned}
right_0 &= store(right_0, null, null) \\
h_0 &= \langle right_0, 0 \rangle \\
O(\langle right, clock \rangle, d, l, r) &= \begin{pmatrix} let\ clock' = clock + 1 \\ let\ o = O(clock', d, l) \\ (\langle store(right, o, r), clock' \rangle, o) \end{pmatrix} \\
data(h, O(id, d, l)) &= d \\
left(h, O(id, d, l)) &= l \\
left(h, null) &= null \\
right(\langle right, clock \rangle, o) &= select(right, o) \\
update_right(\langle right, clock \rangle, o, r) &= \langle store(right, o, r), clock \rangle
\end{aligned}$$

5.3 Not All Encodings are Equal

The advantage of using the built-in algebraic data-types becomes highly visible when the heap gets updated multiple times. For example, in one test we created 1000 objects and then verified that the left child of the first object remained unchanged after the 1000 updates. It takes Z3 18 seconds to instantiate 600,000+ array axioms and establish the equality using the array-based encoding. The second encoding can prove the same theorem in a small fraction of a second. Establishing (11) requires also about 18 seconds and 232,582 quantifier instantiations using the first encoding, and is establish instantaneously using the second encoding.

6 HOL

Sattalax [4] is a theorem prover for Church's Higher-Order Logic (HOL) [1] that is based on simple type theory with Hilbert's choice operator. It won the CASC division for higher-order logic in 2011. The main idea in Sattalax is to reduce problems in HOL to a sequence of SAT problems. Sattalax uses the MiniSAT SAT solver. This, apparent unsophisticated method, has an edge over current competing tools thanks to the highly tuned SAT solver MiniSAT, and a judicious combination of strategies in Sattalax. The Sattalax reduction uses several components: It searches for quantifier instances for quantified formulas. It then encodes satisfiability of quantifier-free formulas into propositional logic.

The purpose of this section is very straight-forward. It is to show how to leverage an SMT solver for handling the encoding of ground formulas into propositional logic. The other much more profound challenge remains, and we don't address it here: sophistication and tuning for finding useful quantifier instantiations.

There is a set of variables $\mathcal{V}$ with elements $x, y, z, \ldots$. The theory HOL is based on simply typed λ calculus. It includes a special sort o of propositions and i of individuals. Types are of the form:

$$\sigma ::= i \mid o \qquad \tau ::= \sigma \mid \tau \to \tau$$

Furthermore, we use the notation $\overline{\tau}$ as a shorthand for $\tau_1, \ldots, \tau_n$ and $\overline{\tau} \to \sigma$ as a shorthand for $\tau_1 \to \ldots \to \tau_n \to \sigma$. Terms are of the form:

$$M, N ::= \lambda x : \tau \ . \ M \mid (M \ N) \mid x$$

We assume also a fixed set of interpreted constants:

$$\begin{array}{l} \mathit{false} : o, \quad \Longrightarrow : o \to o \to o, \\ \epsilon : (\tau \to o) \to \tau, \quad \forall : (\tau \to o) \to o, \quad = : \tau \to \tau \to o \qquad \text{for each type } \tau \end{array}$$

As usual, terms are assumed simply and well-typed: $(M \ N)$ can only be formed if M has type $\tau \to \tau'$ and N has type τ. We write M^τ for a term M with type τ (under a type environment Γ). Simply typed terms are strongly normalizing, so they admit $\beta\eta$ normal forms that we denote $M\downarrow$. Equality under α-renaming can be dealt with by using de-Bruijn indices. HOL is generally highly incomplete (it can encode Peano arithmetic) but it is complete under Henkin [12] semantics. Under the Henkin term-based semantics the set of values in every type τ comprises of the all the closed terms of type τ. This set is non-empty for every τ because we can always include $\epsilon(\lambda x : \tau.\mathit{false})$. The interpreted constants are characterized by

$$(\forall \ (\lambda x : \tau \ .\neg(M \ x))) \ \vee \ (M \ (\epsilon \ M)) \qquad \text{for every } M : \tau \to o \tag{2}$$

$$M = N \Leftrightarrow (\forall \lambda x : \tau \ . \ (M \ x) = (N \ x)) \qquad \text{for every } M, N : \tau \to \tau' \tag{3}$$

$$(\forall \ M) \Longrightarrow (M \ N) \qquad \text{for every } M : \tau \to o, N : \tau \tag{4}$$

together with the usual congruence properties of equality and the Boolean connectives $\Longrightarrow$ and *false* (and the definitions for derived abbreviations $\neg, \Leftrightarrow, \vee$, and $\wedge$). Furthermore, $M\downarrow = M$ for every M. The main idea of Sattalax is to saturate a goal F under these properties. Since Sattalax is based on a SAT solver it also has to saturate with respect to the theory of equality. The main point made here is that this part of the theory propagation is already taken care of by SMT solvers that provide ground equality reasoning as a built-in feature. Saturation causes the properties to be instantiated by every constructable terms M, N. Two challenges arise, the first is to find a way to enumerate all constructable terms, the second is to enumerate the terms in an order that is useful for finding short proofs. In general, one must fairly enumerate every type τ and every term of type τ. We sketch a construction of sets of terms with free variables from the

typing context Γ and of type τ as the set $\mathcal{T}[\Gamma;\tau]$. It is the least fixed-point under the membership constraints:

$$\begin{array}{lll}(\lambda x:\tau\ .\ M)\in\mathcal{T}[\Gamma;\tau\to\tau'] & \text{if} & M\in\mathcal{T}[\Gamma,x:\tau;\tau']\\ (x\ M_1\ \ldots\ M_k)\in\mathcal{T}[\Gamma;\sigma] & \text{if} & (x:\overline{\tau}\to\sigma)\in\Gamma,\ M_i\in\mathcal{T}[\Gamma;\tau_i]\end{array}$$

The constructed terms are in $\beta\eta$ long normal form. We here assume that Γ is pre-populated with the constants *false* and $\Longrightarrow$ and for every type τ a corresponding instance of $\forall,\epsilon,=$. A useful approach for enumerating the terms is to fix a depth towards the number of times one is willing to use either of the saturation rules above and then enumerate all terms and types up to the fixed depth.

6.1 Leveraging Theories

The translation of λ-terms into first-order terms can exploit the support for equality and propositional logic that already exists in the context of SMT solvers. We give the translation function $[\![\_]\!]$ to the right. It creates quoted terms $\lceil M\rceil$ for λ-terms that don't correspond to Z3-expressible terms. The quoted terms are treated as uninterpreted constants from Z3's point of view. The theory of *extensional* arrays furthermore lets us enforce that application is extensional without having to expand axioms for extensionality ourselves. In other words, the function select satisfies $(\forall x:\tau\ .\ \mathsf{select}(M,x)=\mathsf{select}(N,x))\Longrightarrow M=N$. We can therefore replace (3) with only the left-to-right implication.

$$\begin{aligned}[\![(\forall\ M)]\!] &= \lceil(\forall\ M)\rceil\\ [\![(\epsilon\ M)]\!] &= \lceil(\epsilon\ M)\rceil\\ [\![M\Longrightarrow N]\!] &= [\![M]\!]\Longrightarrow[\![N]\!]\\ [\![M=N]\!] &= [\![M]\!]=[\![N]\!]\\ [\![(M\ N)]\!] &= \mathsf{select}([\![M]\!],[\![N]\!])\\ [\![\lambda x:\tau\ .\ M]\!] &= \lceil\lambda x:\tau\ .\ M\rceil\\ [\![f]\!] &= f\quad\text{for constant } f\end{aligned}$$

We are now ready to outline the basic saturation loop for HOL. Initialize the depth $d\leftarrow 0$. Assert $[\![F\downarrow]\!]$. Then repeatedly apply the following steps until $[\![F\downarrow]\!]$ is ground unsatisfiable:

1. F contains the sub-term $\lceil(\epsilon\ M)\rceil$, then add $[\![(2)\downarrow]\!]$ to F.
2. F contains the sub-term $[\![M^{\tau\to\tau'}=N]\!]$, then add $[\![(3)\downarrow]\!]$ to F.
3. F contains the sub-term $\lceil(\forall\ M^{\tau\to o})\rceil$, then for every $N\in\mathcal{T}[\epsilon;\tau]$ of depth less than d add $[\![(4)\downarrow]\!]$ to F.
4. $d\leftarrow d+1$.

Remark 2. We can in principle retain even more of the structure of λ-terms when interpreting them in the context of Z3. The support for the theory of arrays [6] in Z3 includes native handling of combinators $K:\tau\to(\tau'\Rightarrow\tau)$ (the constant array), and $map:(\tau\Rightarrow\tau')\to(\tau''\Rightarrow\tau)\to(\tau''\Rightarrow\tau')$ (a map combinator), besides the function $\mathsf{store}:(\tau\Rightarrow\tau')\to\tau\to\tau'\to(\tau\Rightarrow\tau')$ that updates an array at a given index. The ground theory with these combinators is decidable (satisfiability is NP complete). We call the theory CAL for combinatory array logic. We could therefore in principle extend $[\![\_]\!]$ with the cases $[\![\lambda x\ .\ M]\!]=(K\ [\![M]\!])$ if $x\notin FV(M)$, and $[\![\lambda x\ .\ (M\ (N\ x))]\!]=map([\![M]\!],[\![N]\!])$ when $x\notin FV(M)\cup FV(N)$. It would be interesting to explore to which extent CAL can

be leveraged for solving HOL formulas. We could for instance prove $f \circ g = g \circ f \Rightarrow f \circ g \circ g = g \circ g \circ f$ using the decision procedure for CAL.

We implemented a light-weight HOL theorem prover based on the presented method using Z3. It is not tuned, but can (given some patience) for instance prove that injective functions have inverses: $(\forall x, y : i \;.\; (fx) = (fy)) \Longrightarrow \exists g : i \to i \;.\; \forall x : i \;.\; (g\ (f\ x)) = x$ by synthesizing the instantiation $g := \lambda x : i \;.\; (\epsilon\ (\lambda y : i \;.\; (f\ y) = x))$.

7 Conclusions

We examined a number of theories. The theories were not native to Z3, but could be either encoded using existing theories, be supported by saturating with theory axioms, or be supported efficiently using custom solvers that work in tandem with core solver. Other constraint satisfiability problems can be encoded as custom theory solvers. This includes both thoroughly and partially explored applications, such as custom constraint propagators for scheduling domains, theories with transitive closure and fixed-point operators, local theory extensions, separation logic and answer set programming.

Thanks to Chris Brown, Albert Oliveras, Nikolai Tillmann, Andrei Voronkov and Matt Dwyer for their inspiration and input on the theories and examples used here.

References

1. Church, A.: A Formulation of the Simple Theory of Types. Journal of Symbolic Logic 5, 56–68 (1940)
2. Banerjee, A., Naumann, D., Rosenberg, S.: Decision Procedures for Region Logic. In: Submission (August 2011), http://www.cs.stevens.edu/~naumann/publications/dprlSubm.pdf
3. Bjørner, N.: Linear quantifier elimination as an abstract decision procedure. In: Giesl and Hähnle [9], pp. 316–330
4. Brown, C.E.: Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 147–161. Springer, Heidelberg (2011)
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The Opensmt Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
6. de Moura, L., Bjørner, N.: Efficient, Generalized Array Decision Procedures. In: FMCAD. IEEE (2009)
7. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (2011)
9. Giesl, J., Hähnle, R. (eds.): IJCAR 2010. LNCS, vol. 6173, pp. 91–106. Springer, Heidelberg (2010)

10. Gil, J., Zibin, Y.: Efficient dynamic dispatching with type slicing. ACM Trans. Program. Lang. Syst. 30(1) (2007)
11. Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., Levin, M.Y.: Automating Software Testing Using Program Analysis. IEEE Software 25(5), 30–37 (2008)
12. Hoder, K., Kovács, L., Voronkov, A.: Interpolation and symbol elimination in vampire. In: Giesl and Hähnle [9], pp. 188–195
13. Henkin, L.: Completeness in the theory of types. Journal of Symbolic Logic 15, 81–91 (1950)
14. Lahiri, S.K., Seshia, S.A., Bryant, R.E.: Modeling and Verification of Out-of-Order Microprocessors in Uclid. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 142–159. Springer, Heidelberg (2002)
15. Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
16. Rümmer, P., Wintersteiger, C.: Floating-point support for the Z3 SMT Solver, http://www.cprover.org/SMT-LIB-Float
17. Sherman, E., Garvin, B.J., Dwyer, M.B.: A slice-based decision procedure for type-based partial orders. In: Giesl and Hähnle [9], pp. 156–170
18. Suter, P., Steiger, R., Kuncak, V.: Sets with Cardinality Constraints in Satisfiability Modulo Theories. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 403–418. Springer, Heidelberg (2011)

Algebra, Logic, Locality, Concurrency

Peter W. O'Hearn

Queen Mary University of London

This talk reports on ongoing work – with Tony Hoare, Akbar Hussain, Bernhard Möller, Rasmus Petersen, Georg Struth, Ian Wehrman, and others – on models and logics for concurrent processes [10,6,5]. The approach we are taking abstracts from syntax or particular models. Message passing and shared memory process interaction, and strong (interleaving) and weak (partial order) approaches to sequencing, are accomodated as different models of the same core axioms. Rules of program logic, related to Hoare and Separation logics, flow at once from the algebraic axioms. So, one gets a generic program logic from the algebra, which holds for a range of concrete models.

The most notable amongst the algebra laws is an ordered cousin of the exchange law of 2-categories or bicategories, which here links primitives for sequential and parallel composition

$$(p \parallel r); (q \parallel s) \ \sqsubseteq \ (p; q) \parallel (r; s).$$

This law was was noticed in work on pomsets and traces in the 1980s and 1990s [4,1], and emphasized recently in the formulation of Concurrent Kleene Algebra [5]. An important observation of [5] is that by viewing the pre/post spec $\{p\}\, c\, \{q\}$ as a certain relation in the algebra – there are actually two such, $p; c \sqsubseteq q$ and $c; q \sqsubseteq p$ – one obtains a number of rules for program logic. The use of ; to separate the precondition and program, or program and postcondition, has an interesting consequence: if the sequential composition is a 'weak' one that allows statement re-ordering (as in weak or relaxed memory models that do not guarantee sequentially consistent behaviour, or more generally as available in partial order models such as pomsets or event structures [11,9]) then we still obtain rules of sequential Hoare logic. And when combined with $\parallel$ using the exchange law, it results in very general versions of the rules

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * P_2\}\, C_1 \| C_2\, \{Q_1 * Q_2\}}\ \texttt{Concurrency} \qquad \frac{\{P\}\, C\, \{Q\}}{\{P * F\}\, C\, \{Q * F\}}\ \texttt{Frame}$$

which in Concurrent Separation Logic support modular reasoning about concurrent processes [7], where $*$ is the separating conjunction (which holds when its conjuncts holds of separate resources).

A remarkable fact is that the initial conception of these rules from Concurrent Separation Logic is strongly based on an idea of 'locality of resource access' [8,2,3], where such intuitions do not seem to be present in the algebraic theory. For instance, in the frame rule we understand that $\{P\}\, C\, \{Q\}$ implies that command C only accesses those resources described by precondition P, and this justifies tacking on a description of separate resources that will thus not

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 17–18, 2011.

be altered (the $*F$ part). Similarly, in the concurrency rule we understand that processes started in separate states will not trample on one another's resources, because of locality. The notion of 'locality of resource access' is a semantic notion that underlies the semantics of Separation Logic: the soundness of the Frame and Concurrency has been proven by validating properties of the semantics of programs that express locality of resource access (properties which incidentally are independent of the syntax of the logic) [12,3]. However, such forms of justification are not needed at all in the algebra.

The understanding of this point – how locality and the algebra are related – is a particular focus of the talk. We start from a standard model of resources, and construct an algebra from it, making a link between the intuitions concerning locality of resource access and the axioms in the algebra. Perhaps surprisingly, the algebra is seen to contain a general account of locality, which strictly generalizes the modular reasoning of Concurrent Separation Logic [5].

On the other hand, the algebra has as instances concrete models that are far removed conceptually from the resource models at the basis of Separation Logic (e.g., models based on interleaving and independence of events), and this leads to the question of whether it is possible to uniformly obtain effective modular reasoning techniques for a wide range of models of concurrency.

References

1. Bloom, S.L., Ésik, Z.: Free shuffle algebras in language varieties. Theor. Comput. Sci. 163(1&2), 55–98 (1996)
2. Brookes, S.D.: A semantics of concurrent separation logic. Theoretical Computer Science 375(1-3), 227–270 (2007); Prelim. version appeared in CONCUR 2004
3. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS, pp. 366–378. IEEE Computer Society (2007)
4. Gischer, J.L.: The equational theory of pomsets. Theor. Comput. Sci. 61, 199–224 (1988)
5. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On Locality and the Exchange Law for Concurrent Processes. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. LNCS, vol. 6901, pp. 250–264. Springer, Heidelberg (2011)
6. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program (2011); Preliminary verson in CONCUR 2009
7. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007); Prelim. version appeared in CONCUR 2004
8. 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, pp. 1–19. Springer, Heidelberg (2001)
9. Pratt, V.: Modelling concurrency with partial orders. International Journal of Parallel Programming 15(1), 33–71 (1986)
10. Wehrman, I., Hoare, C.A.R., O'Hearn, P.W.: Graphical models of separation logic. Inf. Process. Lett. 109(17), 1001–1004 (2009)
11. Winskel, G.: Events in Computation. Ph.D. thesis, University of Edinburgh (1980)
12. Yang, H., O'Hearn, P.W.: A Semantic Basis for Local Reasoning. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, p. 402. Springer, Heidelberg (2002)

Modular Abstractions of Reactive Nodes Using Disjunctive Invariants*

David Monniaux[1] and Martin Bodin[1,2]

[1] CNRS / Verimag
[2] École normale supérieure de Lyon

Abstract. We wish to abstract nodes in a reactive programming language, such as Lustre, into nodes with a simpler control structure, with a bound on the number of control states. In order to do so, we compute disjunctive invariants in predicate abstraction, with a bounded number of disjuncts, then we abstract the node, each disjunct representing an abstract state. The computation of the disjunctive invariant is performed by a form of quantifier elimination expressed using SMT-solving.

The same method can also be used to obtain disjunctive loop invariants.

1 Introduction

Our goal is to be able to compute sound abstractions of reactive nodes, with tunable precision. A reactive node in a language such as Lustre,[1] or Scade,[2] Sao,[3] or even Simulink,[4] has input streams, output streams, and an (optional) internal state: at each clock cycle, the value on each output is a function of the values on the inputs and the state; and so is the next value of the state.

If the state consists in a finite vector of Booleans, or other finite values, then the node is a finite automaton, with transitions guarded according to the current values of the inputs, and for each state a relation between the current values of the inputs and the current values of the outputs. This is often referred to as the *control structure* of the reactive program. The problem with that representation, which exposes the full internal state, is that the number of states grows exponentially with the number of state variables, making it unwieldy for analysis.

* This work was partially supported by ANR project "ASOPT".

[1] Lustre is a synchronous programming language, which gets compiled into C. [2].

[2] Scade is a graphical synchronous programming language derived from Lustre. It is available from Esterel Technologies. It has been used, for instance, for implementing parts of the Airbus A380 fly-by-wire systems.

[3] Sao is an earlier industrial graphical synchronous programming language. It has been used, for instance, for implementing parts of the Airbus A340 fly-by-wire systems.

[4] Simulink is a graphical data-flow modeling tool sold as an extension to the Matlab numerical computation package. It allows modeling a physical or electrical environment along the computerized control system. A code generator tool can then provide executable code for the control system for a variety of targets, including generic C. Simulink is available from The Mathworks.

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 19–33, 2011.

The problem is even more severe if the control conditions are not directly exposed as Boolean state variables, but as predicates over, say, integer or real variables (see example in Sec. 4).

The main contribution of this article is a method for constructing a more abstract automaton, with a bounded number of states ($\leq n$), whose behaviors still over-approximate the behaviors of the node. In order to do so:

1. We compute an over-approximation of the set of reachable states of the node, in an unspecified context, as a union of at most n "abstract states", each defined by a conjunction of constraints (these abstract states need not be disjoint).
2. We compute the most precise transition relation between these abstract states.

This automatic abstraction maps a reactive node into another, more abstract (and, in general, nondeterministic) reactive node. This enables modular and compositional analysis: if a node is composed of several nodes, then one can replace each of these nodes by its abstraction, and then analyze the compound node.

As a secondary contribution, the analysis method at step 1 can also be used to obtain disjunctive loop invariants for imperative programs (or, more generally, invariants for arbitrary control flow graphs), given a precondition and an optional postcondition. We describe this algorithm for obtaining invariants in disjunctive normal form, but it in fact also works for other templates.

Our algorithms use *satisfiability modulo theory* (SMT) solving as an essential subroutine; see e.g. [3] for an introduction.

2 Invariants by Predicate Abstraction

Predicate abstraction abstracts program states using the truth value of a given finite set of predicates $\{\pi_1, \ldots, \pi_m\}$: each state σ is abstracted by a m-tuple of Booleans$(\pi_1(\sigma), \ldots, \pi_m(\sigma))$. The most precise abstract transition relation between such vectors of Booleans is $(B_1, \ldots, B_m) \rightarrow_\pi (B'_1, \ldots, B'_m)$ if and only if there exist $\sigma \models \bigwedge(\pi_i = B_i)$, $\sigma' \models \bigwedge(\pi_i = B'_i)$, and $\sigma \rightarrow \sigma'$ where $\rightarrow$ is the transition relation of the program. Then, given an abstract initial state, the set of reachable states of the abstract transition relation can be computed within finite time (in general, exponential in m) by Kleene iterations (equivalently, by computing the transitive closure of $\rightarrow_\pi$).

Such an approach is, however, unworkable in general because of the exponential number of states generated, and thus all current predicate abstraction schemes use some stronger form of abstraction [7]; for instance, they may simply compute a conjunction of the π_i that holds inductively at a given program point. Conjunctive invariants are however fairly restrictive; in this article, we consider the problem of obtaining invariants as *disjunctions of a fixed number of conjunctions* of the chosen predicates.

The set of reachable states of a reactive node, in an unspecified environment, is the strongest invariant of an infinite loop:

```
while (true) {
  i = inputs();
  o = outputs(state, i);
  state = next_state(state, i);
}
```

We shall therefore investigate the problem of automatically finding disjunctive inductive loop invariants (or, more generally, invariants for predicate abstraction following a fixed template), using predicate abstraction, given a precondition and an optional postcondition. These invariants shall be minimal with respect to the inclusion ordering: there shall be no stronger inductive invariant definable by the same template.

2.1 Solution of a Universally Quantified Formula

Let us assume a finite set $\Pi = \{\pi_1, \ldots, \pi_m\}$ of predicates over the state space of the variables of the program. Let $n \geq 1$ be an integer. We are looking for invariants of the form $C_1 \vee \cdots \vee C_n$ where the C_i are conjunctions of predicates from Π (most of our techniques are not specific to this template form, see Sec. 2.5 for extensions).

Any such invariant can be obtained by instantiating the Booleans $b_{i,j}$ in the following template:

$$\mathcal{T} \triangleq \bigvee_i \underbrace{\bigwedge b_{i,j} \Rightarrow \pi_j}_{C_i} \tag{1}$$

Setting $b_{i,j}$ to true(respectively, false) in that template means that predicate π_j appears (respectively, does not appear) in the i-th disjunct C_i. For instance, if $\Pi = \{x > 0, x < 1, y > 0\}$ and $n = 2$, then $b_{1,1} = \mathsf{true}$, $b_{1,2} = \mathsf{true}$, $b_{1,3} = \mathsf{false}$, $b_{2,1} = \mathsf{false}$, $b_{2,2} = \mathsf{false}$, $b_{2,3} = \mathsf{true}$ correspond to $(x > 0 \wedge x < 1) \vee y > 0$.

The problem of finding an invariant reduces to finding suitable values for these Booleans. There is therefore a search space for invariant candidates of *a priori* size 2^{mn}. We impose that the invariant I obtained be *minimal* within that search space with respect to the inclusion ordering; that is, there is no I' expressive using the template such that $I' \subsetneq I$.

Our algorithm can in fact apply to any control-flow graph. For the sake of simplicity, we shall describe it on a single loop.

In Hoare logic, the conditions for proving that a postcondition P holds after a while loop whose condition is C, whose transition relation is T and whose precondition is S using loop invariant I are:

- I must contain the precondition, otherwise said $\forall \sigma\ S \Rightarrow I$.
- I must be inductive, otherwise said $\forall \sigma, \sigma'\ I \wedge C \wedge T \Rightarrow I'$, with I' denoting I where all state variables have been primed.
- $I \wedge \neg C$ must imply the postcondition, otherwise said $\forall \sigma\ I \wedge \neg C \Rightarrow P$.

If we impose I to be an invariant of the required form, that is, an instantiation $\mathcal{T}[B/b]$ of $\mathcal{T}$ obtained by setting the $b_{i,j}$ variables to certain values $B_{i,j}$, these

conditions boil down to the values $B_{i,j}$ of the $b_{i,j}$ variables must satisfy certain formulas universally quantified over the state σ or on the couple of states σ, σ'.

We now make an additional assumption: the states σ or σ' comprise a fixed, *finite* number of variables[5] expressible in a theory $\mathfrak{T}$ for which there exists a satisfiability testing algorithm, in which the predicates $\pi_1, \ldots, \pi_m$ can be expressed, and which allows propositional variables. Thus, the problem boils down to finding a solution to a conjunction of universally quantified formulas of that theory such that the only free variables are the $b_{i,j}$ Booleans.

In the following sections, lowercase σ and σ' stand for states (thus stand for a finite number of variables in the theory $\mathfrak{T}$), uppercase Σ and Σ' stand for values of these state variables. Similarly, lowercase b stands for the matrix of propositional variables $(b_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$, and uppercase B stands for the matrix of Booleans $(B_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$. $F[B/b]$ thus stands for the formula F where the propositional values b have been replaced by the corresponding Booleans in B, and $F[\Sigma/\sigma]$ stands for the formula F where the state variable σ has been replaced by the state value Σ.

2.2 Naive Algorithm for a Given Postcondition

In this section, we shall explain how to compute an invariant suitable for proving the Hoare triple of a loop, given a precondition, a postcondition (which may be true), a loop condition and a transition relation.

Let us first give an intuition of the algorithm. A universally quantified formula $\forall \sigma F$ with free Boolean variables b can be understood as specifying a potentially infinite number of constraints $F[\Sigma/\sigma]$ over b, where Σ ranges all possible values for σ (in this section, we will lump together σ and σ' as a single σ). The idea is to "discover" such constraints one at a time, when they are violated.

Let us now examine the algorithm in more detail; see Sec. 3 for a complete algorithm run. The H_k sequence of propositional formulas over the b variables will express successive refinements of the constraints during the search of a suitable assignment. Initially, we do not know anything about possible solutions, so we set $H_1 \triangleq \mathsf{true}$.

We start by taking any initial assignment $B^{(1)}$ (since any will satisfy H_1) and check whether $\neg F[B^{(1)}/b]$ is satisfiable, that is, whether one can find suitable values for σ. If it is not, then $B^{(1)} \models \forall \sigma\ F$. If it is satisfiable, with example value Σ_1, we add $F[\Sigma_1/\sigma]$ as a constraint — that is, we take $H_2 \triangleq H_1 \wedge F[\Sigma_1/\sigma]$; note that this constraint excludes $B^{(1)}$ and possibly other values for b. Now find an assignment $B^{(2)}$ satisfying H_2, check whether $\neg F[B^{(2)}/b]$ is satisfiable. If it is not, then $B^{(2)} \models \forall \sigma\ F$. If it is satisfiable, with example value Σ_2, we take $H_3 = H_2 \wedge F[\Sigma_2/\sigma]$; note that H_3 excludes $B^{(1)}$ and $B^{(2)}$. The process continues until a suitable assignment is found or the constraints exclude all assignments. Note that one Boolean assignment at least is excluded at each iteration, and

[5] These variables are not necessarily scalar variables. It is for instance possible to consider uninterpreted functions from the integers to the integers, which stand for a countably infinite number of integers.

that the number of Boolean assignments is finite (exponential in the number of propositional variables in b).

More formally: recall that we have reduced our problem of finding an invariant to finding Boolean values $B_{i,j}$ such that $(B_{i,j})_{1\leq i\leq m,1\leq j\leq n} \models \forall\sigma\ F$ for a certain quantifier-free formula F whose free variables are $(b_{i,j})_{1\leq i\leq m,1\leq j\leq n}$. Let us now assume we have a SMT-solver for theory $\mathfrak{T}$, a function $SMT(G)$ which given a formula G answers $sat(M)$ when G is satisfiable, where M is a model, that is, a suitable instantiation of the free variables in G, or *unsat* otherwise. We shall also assume a SAT-solver SAT with similar notations, for purely propositional formulas. We run the following algorithm, expressed in pseudo-ML:

```
H := true
loop
  match SAT(H) with
  | unsat → return "no solution"
  | sat((B_{i,j})_{1≤i≤m,1≤j≤n}) →
    match SMT(¬F[B/b]) with
    | unsat → return "solution B"
    | sat(Σ) → H := H ∧ F[Σ/σ].
```

This algorithm always terminates, since the main loop iterates over a finite set of size $2^{|b|}$ where $|b| = mn$ is the size of the matrix b of propositional variables: the number of models of the propositional formula H decreases by at least one at each iteration, since model B is excluded by the $F[\Sigma/\sigma]$ condition. The loop invariant is $\forall\sigma\ F \implies H$. This invariant is maintained: whatever we choose for Σ, if $\forall\sigma\ F \implies H$, $\forall\sigma\ F \implies H \wedge F[\Sigma/\sigma]$. If the algorithm answers "no solution" for H, because of the invariant, there is no solution for $\forall\sigma\ F$. If the solution answers "solution B", the "unsat" answer for $SMT(\neg F[B/b])$ guarantees that $B \models \forall\sigma\ F$.

Note the use of two solvers: one SAT for the propositional variables b, and one SMT for the state variables σ (or σ, σ'). The SAT solver is used incrementally: one only adds new constraints. The SMT solver is always used with the same set of predicates, enabling it to cache theory lemmas.

2.3 Performance Improvements

The algorithm in the preceding subsection is sound, complete and terminating. Yet, experiments have shown that it tends to generate useless iterations. One reason is that the system may iterate across instances B that yield the same formula $T[B/b]$ up to a permutation of the C_i disjuncts. Another is that the system may generate empty disjuncts C_i, or more generally disjuncts that are subsumed by the other disjuncts (and are thus useless). We shall explain how to deal with those issues.

Removal of Permutations. We impose that the disjunction $C_1 \vee \cdots \vee C_n$ follows a unique canonical ordering. For this, we impose that the vectors of m

Booleans $(B_{1,j})_{1\leq j\leq m}, \ldots, (B_{n,j})_{1\leq j\leq m}$ are in strict increasing order with respect to the lexicographic ordering $\prec_L$ induced by $\mathsf{false} < \mathsf{true}$. This corresponds to $n-1$ constraints $(b_{i,j})_{1\leq j\leq m} \prec_L (b_{i+1,j})_{1\leq j\leq m}$, each of which can be encoded over the propositional variables $(b_{i,j})$ as formula $L_{i,1}$ defined as follows:

- L_{i,j_0} is a formula whose meaning is that $(b_{i,j})_{j_0\leq j\leq m} \prec_L (b_{i+1,j})_{j_0\leq j\leq m}$
- $L_{i,m+1}$ is false
- L_{i,j_0} for $1 \leq j_0 \leq m$ is defined using L_{i,j_0+1} as follows: $(\neg b_{i,j_0} \wedge b_{i+1,j_0}) \vee ((b_{i,j_0} \Rightarrow b_{i+1,j_0}) \wedge L_{i,j_0+1})$.

All such constraints can be conjoined to the initial value of H.

Removal of Subsumed Disjuncts. We can replace the SAT-solver used to find solutions for $(b_{i,j})$ by a SMT-solver for theory $\mathfrak{T}$, in charge of finding solutions for $(b_{i,j})$ and for some auxiliary variables $\sigma_1, \ldots, \sigma_n$ (we actually shall not care about the actual values of $\sigma_1, \ldots, \sigma_n$). The following constraint expresses that the disjunct C_{i_0} is not subsumed by the disjuncts $(C_i)_{1\leq i\leq n, i\neq i_0}$:

$$\exists \sigma_{i_0}\ C_{i_0}[\sigma_{i_0}/\sigma] \wedge \bigwedge_{1\leq i\leq n, i\neq i_0} \neg C_i[\sigma_i/\sigma] \tag{2}$$

It therefore suffices to conjoin to the initial value of H the following constraints, for $1 \leq i_0 \leq n$: $C_{i_0}[\sigma_{i_0}/\sigma] \wedge \bigwedge_{1\leq i\leq n, i\neq i_0} \neg C_i[\sigma_i/\sigma]$.

A variant consists in simply imposing that each of the C_i is satisfiable, thus eliminating useless false disjuncts. For this, one imposes $1 \leq i_0 \leq n$, the constraint $C_{i_0}[\sigma_{i_0}/\sigma]$. Equivalently, one can pre-compute the "blocking clauses" over the $b_{i_0,j}$ propositional variable that constrain these variables so that C_{i_0} is satisfiable, and add them as purely propositional constraint. This is the method that we used for the example in Sec. 3 (we wanted to keep to propositional constraints for the sake of simplicity of exposition).

2.4 Iterative Refinement of Invariants

We have so far explained how to compute *any* invariant, with or without imposing a postcondition. If we do not impose a postcondition, the formula true, for instance, can denote a wholly uninteresting invariant; clearly we would like a smaller one. In this section, we shall explain how to obtain *minimal* invariants within the search space.

For a Fixed Disjunction Size. Let us now assume we have the postcondition P (if we do not have it, then set P to true). A natural question is whether one can get a *minimal* inductive invariant of the prescribed form for the inclusion ordering; that is, an invariant $T[B_0/b]$ such that there exists no B such that $T[B/b] \subseteq T[B_0/b]$, by which we denote $\forall \sigma\ T[B/b] \Rightarrow T[B_0/b]$. We shall now describe an iterative algorithm that first obtains any inductive invariant of the prescribed form, and then performs a downwards iteration sequence for the inclusion ordering, until a minimal element is found.

Let us first note that it is in general hopeless to find a global minimum B_0, that is, one such that $\forall B\ T[B_0/b] \subseteq T[B/b]$, for there may exist incomparable minimal elements. For instance, consider the program:

```
float i = 0;
while(random()) {
  i = i+1;
  if (i > 2) i = 0;
}
```

The least inductive invariant of this loop, for variable i, is the set of floating-point numbers $\{0, 1, 2\}$. Now assume our set of predicates is $\{i \leq 0, i \geq 0, i \geq 1, i \leq 1, i \leq 2, i \geq 2\}$, and take $n = 2$; we thus look for disjunctions of two intervals. Two minimal incomparable invariants are $(i \geq 0 \wedge i \leq 1) \vee (i \geq 2 \wedge i \leq 2)$, that is, $[0, 1] \cup \{2\}$, and $(i \geq 1 \wedge i \leq 2) \vee (i \leq 0 \wedge i \geq 0)$, that is, $[1, 2] \cup \{0\}$.

Let us now assume we have already obtained an invariant $T[B'/b]$ and we wish to obtain a better invariant $T[B/b] \subsetneq T[B'/b]$. This last constraint can be written as the conjunction of:

1. $T[B/b] \subseteq T[B'/b]$, otherwise said $\forall\sigma\ T[B/b] \Rightarrow T[B'/b]$; such a universally quantified constraint can be handled as explained in Sec. 2.2.
2. $\exists\sigma\ T[B'/b] \wedge \neg T[B/b]$. Again, as explained in Sec. 2.3, one can treat such an existentially quantified constraint by using a SMT-solver instead of a SAT-solver and adding to H an extra variable σ and the constraint $T[B'/b] \wedge \neg T[B/b]$. When an invariant $T[B/b]$ is found, the value Σ of σ is a witness that this invariant is *strictly included* in $T[B'/b]$.

It is possible to compute a downward iteration sequence until a minimal element is reached: compute any initial invariant $B^{(0)}$, then $B^{(1)} \subsetneq B^{(0)}$ etc. until the system fails to provide a new invariant satisfying the constraints; one then takes the last element of the sequence. The termination condition is necessarily reached, for the $(B^{(k)}_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$ Boolean matrices can never be twice the same within the sequence (because of the strict descending property). Furthermore, one can stop at any point $B^{(k)}$ within the sequence and get a (possibly non minimal) inductive invariant.

One can replace point 2 above by a weaker strategy, but with the advantage of operating only on propositional formulas. Note that $B^{(k+1)}$ has at least one component higher than $B^{(k)}$ for the standard ordering $\mathsf{false} < \mathsf{true}$ on the Booleans, for if all components are lower or equal, then $B^{(k+1)} \supseteq B^{(k)}$, which is the opposite direction of what we wish. The strategy is to enforce this condition using $\bigvee_{i,j}(b_{i,j} \wedge \neg b'_{i,j})$. This is what we used in Sec. 3.

For Varying Disjunction Sizes. The algorithms described above work for a given disjunction size n. The method for preventing subsumed disjuncts of part Sec. 2.3 imposes that all n disjuncts are truly needed: it is thus possible that no solution should be found for $n = n_0$ while solutions exist for $n = n_0 - 1$.

We therefore suggest that, once a minimal invariant I_{n_0} is obtained for $n = n_0$ fixed, one looks for an invariant strictly included in I_{n_0} for $n = n_0 + 1$. One can

choose to stop such iterations when no solutions are found for a given n, or when a limit on n or a timeout is reached.

2.5 Extensions

Prohibition of Overlapping Modes. Our algorithms produce disjunctions that cover all reachable states, but that do not define partitions: distinct abstract states may be overlapping. This may be somewhat surprising and counterintuitive.

It is possible to impose that disjuncts should be pairwise disjoint. For any i and j, one can impose that C_i and C_j are disjoint by the universally quantified formula $\forall\sigma\neg C_i \vee \neg C_j$. We have explained in the preceding sections how to deal with such universally quantified formulas.

Other Template Forms. We have described our algorithm for templates of the form $C_1 \vee \cdots \vee C_m$ where the C_i are conjunctions constructed from the chosen predicates, but the algorithm is not specific to this template shape. Instead of disjunctive normal form, one could choose conjunctive normal form, for instance, or actually any form [23], though reductions of the search space such as those from Sec. 2.3 or 2.3 may be more difficult to define.

Predicate Choice. Our method is based on predicate abstraction; so far we have not discussed methods for obtaining the predicates, beyond the obvious syntactic detection. In many systems based on predicate abstraction, one uses *counterexample-based abstraction refinement* (CEGAR): from an abstract trace violating the specification, but not corresponding to a concrete trace violating the specification, one derives additional predicates for refining the system. Because we did not implement such refinement, we shall only give a rough description of our CEGAR method.

If there is no inductive invariant built from the requested template that can prove the desired postcondition, the algorithm from Sec. 2.2 will end up with an unsatisfiable constraint system. This system is unsatisfiable because of the postcondition constraints (otherwise, in the worst case, one would obtain a solution yielding the `true` formula); relevant postcondition constraints can be obtained from an unsatisfiable core of the constraint system. One can then try removing such constraints one by one until the constraint system becomes satisfiable again. Any solution of this relaxed constraint system defines an inductive invariant, but one that does not satisfy the postcondition. As with the usual CEGAR approach, one could try generating test traces leading from the initial states to the complement of the postcondition and staying within the invariant; if the postcondition holds, such searches are unsuccessful and yield interpolants from which predicates may be mined.

3 Step-by-Step Example of Invariant Inference

For the sake of simplicity of exposition, in this section we have restricted ourselves to pure propositional constraints on the $b_{i,j}$, and satisfiability modulo the

theory of linear integer arithmetic for the combination of the $b_{i,j}$ and the state variables. We consider the following simple program.

```
int b, i=0, a; /* precondition a > 0 */
while (i < a) {
  b = random();
  if (b)
    i = i + 1;
}
```

The predicates are $\{\pi_1, \ldots, \pi_8\} \triangleq \{i = 0, i < 0, i > 0, i = a, i < a, i > a, b, \neg b\}$. The state variable σ stands for (i, a, b). For the sake of simplicity, we model i and a as integers in $\mathbb{Z}$, and b as a Boolean. We assume the loop precondition $S \triangleq i = 0 \wedge a \geq 1$. The loop condition is $C \triangleq i < a$, and the transition relation is $T \triangleq (b' \wedge i' = i + 1) \vee (\neg b' \wedge i' = i)$. We choose $n = 2$.

We shall now run the algorithm described in Sec. 2.2 with the iterative refinement of Sec. 2.4. For the sake of simplicity, we shall use none of the improvements described in the preceding sections that need the H_i to contain non propositional variables: no removal of subsumed disjuncts as described in Sec. 2.3 and no strict inclusion enforcement as described in Sec. 2.4.

We initialize H as follows: H_1 contains Boolean constraints on $(b_{i,j})_{1 \leq i \leq 2, 1 \leq j \leq 8}$

- That prevent C_1 and C_2 from being unsatisfiable, using blocking clauses as explained in Sec. 2.3: one cannot have both $i = 0$ and $i > 0$, and so on.
- That force $(b_{1,j})_{1 \leq j \leq 8} \prec_L (b_{2,j})_{1 \leq j \leq 8}$ for the lexicographic ordering $\prec_L$ on Boolean vectors (this avoids getting the same disjunction twice with the disjuncts swapped).

Let us now see the constraint solving and minimization steps.

1. We perform SAT-solving on H_1 and obtain a satisfying assignment $B^{(1)}_{1,1} = \mathsf{true}, B^{(1)}_{1,2} = \mathsf{false}, B^{(1)}_{1,3} = \mathsf{false}, B^{(1)}_{1,4} = \mathsf{true}, B^{(1)}_{1,5} = \mathsf{false}, B^{(1)}_{1,6} = \mathsf{false}, B^{(1)}_{1,7} = \mathsf{true}, B^{(1)}_{1,8} = \mathsf{false}, B^{(1)}_{2,1} = \mathsf{true}, B^{(1)}_{2,2} = \mathsf{false}, B^{(1)}_{2,3} = \mathsf{false}, B^{(1)}_{2,4} = \mathsf{true}, B^{(1)}_{2,5} = \mathsf{false}, B^{(1)}_{2,6} = \mathsf{false}, B^{(1)}_{2,7} = \mathsf{false}, B^{(1)}_{2,8} = \mathsf{true}$. This corresponds to the invariant-candidate $T[B^{(1)}/b]$, that is, $(i = 0 \wedge i = a \wedge b) \vee (i = 0 \wedge i = a \wedge \neg b)$.

   Now is this invariant-candidate truly an inductive invariant? It is not, because it does not contain the whole of the set of initial states. SMT-solving on $S \wedge \neg T[B^{(1)}/b]$ gives a solution $\Sigma_1 \triangleq (i = 0, a = 1, b = \mathsf{false})$. We therefore take $H_2 \triangleq H_1 \wedge F[\Sigma_1/\sigma]$.
2. A satisfying assignment $B^{(2)}$ of H_2 yields the invariant candidate $(i = 0 \wedge i = a \wedge b) \vee (i = 0 \wedge i < a \wedge b)$. Again, SMT-solving shows this is not an invariant because it does not contain the initial state $\Sigma_2 \triangleq (i = 0, a = -1, b = \mathsf{false})$. We therefore take $H_3 \triangleq H_2 \wedge F[\Sigma_2/\sigma]$.
3. A satisfying assignment $B^{(3)}$ of H_3 yields the invariant candidate $(i = 0 \wedge i = a \wedge b) \vee (i = 0 \wedge i < a)$. SMT-solving shows this is not inductive, since it is

not stable by the transition $\Sigma_3 \triangleq (i = 0, a = 1, b = \mathsf{false}, i' = 1, b' = \mathsf{true})$. We therefore take $H_4 \triangleq H_3 \wedge F[\Sigma_3/\sigma]$.

4. A satisfying assignment $B^{(4)}$ of H_4 yields the invariant candidate $(i = 0 \wedge i < a \wedge \neg b) \vee b$. SMT-solving shows this is not inductive, since it is not stable by the transition $\Sigma_4 \triangleq (i = 1, a = 3, b = \mathsf{true}, i' = 1, b' = \mathsf{false})$. We therefore take $H_5 \triangleq H_4 \wedge F[\Sigma_4/\sigma]$.
5. A satisfying assignment $B^{(5)}$ of H_5 yields the invariant candidate $(i = 0 \wedge i < a) \vee (i > 0 \wedge i = a \wedge b)$. SMT-solving shows this is not inductive, since it is not stable by the transition $\Sigma_5 \triangleq (i = 0, a = 2, b = \mathsf{false}, i' = 1, b' = \mathsf{false})$. We therefore take $H_6 \triangleq H_5 \wedge F[\Sigma_5/\sigma]$.
6. A satisfying assignment $B^{(6)}$ of H_6 yields the invariant candidate $I_1 \triangleq (i = 0 \wedge i < a) \vee i > 0$. SMT-solving shows this is an inductive invariant, which we retain. We however would like a *minimal* inductive invariant within our search space. As described at the end in Sec. 2.4, we take H_7 the conjunction of H_6 and a propositional formula forcing at least one of the $b_{i,j}$ to be true while $B^{(6)}_{i,j}$ is false. Furthermore, as described in point 1 of Sec. 2.4, we now consider $F_2 \triangleq F \wedge (T \Rightarrow I_1)$, which ensures that we shall from now on only consider invariants included in I_1.
7. A satisfying assignment $B^{(7)}$ of H_7 yields the invariant candidate $(i > 0 \wedge i = a \wedge b) \vee i < a$. SMT-solving shows this is not included in I_1, using $\Sigma_7 \triangleq (i = -47, a = 181, b = \mathsf{true})$. We therefore take $H_8 \triangleq H_7 \wedge F_2[\Sigma_7/\sigma]$.
8. H_8 has no solution. I_1 is thus minimal and the algorithm terminates.

A postcondition for this loop is thus $I_1 \wedge \neg(i < a)$, thus $i > 0 \wedge i = a$. Note that the method did not have to know this postcondition in advance in order to prove it.

4 Construction of the Abstract Automaton

We can now assume that the set of reachable states is defined by a formula $I = I_1 \vee \cdots \vee I_n$, with each formula I_i meant to define a state q_i of the abstract automaton.

To each couple of states (q_i, q_j) we wish to attach an input-output relation expressed as a formula $\tau_{i,j}$ with variables $\mathcal{I}$, ranging over the set of possible current values of the inputs and $\mathcal{O}$ over the set of possible current values of the outputs.

Recall that T is a formula expressing the transition relation of the reactive node, over variables $\mathcal{I}$ (inputs), σ (preceding state), σ' (next state) and $\mathcal{O}$ (outputs). Then the most precise transition relation is:

$$\tau_{i,j} \triangleq \exists \sigma, \sigma' \; I_i \wedge I_j[\sigma'/\sigma] \wedge T \tag{3}$$

Any over-approximation of this relation is a sound transition relation for the abstract automaton. If we have a quantifier elimination procedure for the theory

in which T and the I_i are expressed, then we can compute the most precise $\tau_{i,j}$ as a quantifier-free formula; but we can also, if needed, use an approximate quantifier elimination that yields an over-approximation.

Let us consider, as an example, the following Lustre node. It has a single integer input dir and a single integer output out. If dir is nonzero, then it is copied to out; else out decays to zero by one unit per clock cycle:

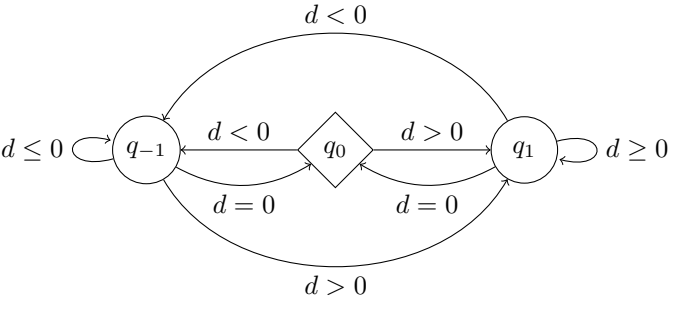
```
node clicker(dir : int) returns (out : int);
let
        out = if dir ≥ 1
              then dir
              else if dir ≤ −1
                   then dir
                   else 0 → if pre out ≤ −1
                            then (pre out) + 1
                            else if pre out ≥ 1
                                 then (pre out) − 1
                                 else 0;
tel.
```

In mathematical notation, let us denote dir by d, **pre** out by o and out by o'. The state consists in a single variable o, thus σ is the same as o. The transition relation then becomes

$$T \triangleq \begin{cases} (d \neq 0 \wedge o' = d) \vee (d = 0 \wedge o \geq 1 \wedge o' = o - 1) \\ \vee (d = 0 \wedge o \leq -1 \wedge o' = o + 1) \vee (d = 0 \wedge o' = o = 0) \end{cases} \tag{4}$$

Suitable predicates are $\{o \leq -1, o = 0, o \geq 1\}$, thus defining the set of reachable states as a partition $I_{-1} \vee I_0 \vee I_1$ where $I_{-1} \triangleq o \leq -1$, $I_0 \triangleq o = 0$, $I_1 \triangleq o \geq 1$.

Let us compute $\tau_{0,1} \triangleq \exists o, o'\ I_0 \wedge I_1[o'/o] \wedge T$, that is, $\exists o, o' o = 0 \wedge o' \geq 1 \wedge T$: we obtain $d > 0$. More generally, by computing $\tau_{i,j}$ for all $i, j \in \{-1, 0, 1\}$, we obtain the automaton below; the initializers (left hand side of the Lustre operator $\rightarrow$) define the initial state q_0.



Note that the resulting automaton is nondeterministic: in state q_1 (respectively, q_{-1}), representing $o > 0$ (resp. $o < 0$), if $d = 0$, then one can either remain in the same state or return to the initial state q_0.

5 Related Work

There have been many approaches proposed for finding invariants and proving properties on transition systems. [21] surveys earlier ones.

The problem of finding the control structure of reactive nodes written in e.g. Lustre has been studied previously, most notably by B. Jeannet [12,13,14], but with respect to a property to prove: the control structure is gradually refined until the property becomes provable. This supposes that we know the desired property in advance, which is not always the case in a modular setting: the property may pertain to another module, and may not be easy to propagate back to the current module. The NBac tool performs such an analysis using convex polyhedra as an abstract domain. More recent methods for refining the control structure of reactive nodes include [1]. We have already proposed some modular abstractions for reactive nodes, but these targeted specific filters with no control structure [15] or needed some precomputation of the control structure [16].

The problem of finding disjunctive invariants has been much studied especially in the context of convex numerical domains, such as polyhedra: if the property to prove is not convex, or relies on a non-convex weakest precondition, then *any* analysis inferring convex invariants will fail. A number of methods have been proposed to infer invariants consisting in finite disjunctions of elements of an abstract domain: some distinguish states according to the history of the computation, as in *trace partitioning* [19], some recombine elements according to some affinity heuristics [20,18], or decompose the transition relation according to some "convexity witness" [10]. Other methods select predicates with which to split the control state [22]. Some recent methods leverage the power of modern SMT-solvers to impose convex invariants only at a limited subset of program points, and distinguish all execution paths between them, therefore acting as applying a complete trace partitioning between the points in the distinguished subset [16,5]; the method in the present article also considers a limited subset of program points (e.g. loop heads), but can infer disjunctive invariants at these points too.

Both polyhedral abstraction and predicate abstraction search for an inductive invariant I; then, in order to prove that a certain property P always holds, one shows that I is included in P. In all static analyzers by abstract interpretation known to the authors, some form of forward analysis is used: the set of initial states influences the invariant I obtained by the system. In contrast, with k-induction, as in the Kind tool [11] the initial states play a very limited role (essentially, they invalidate P if there exists a trace of k states starting in an initial state such that one of them does not satisfy P). A known weakness of pure k-induction is that it may fail to prove a property because it bothers about bad, but unreachable, states. If one has obtained an invariant I by other methods, one can use it to constrain the system and get rid of these bad, unreachable states. Thus, abstraction-based methods and k-induction based methods nicely combine.

The algorithms presented in this article can be seen as a form of minimization constrained by a universally quantified formula $\forall \sigma\ F$, achieved by maintaining

a formula H such that $\forall\sigma\ F \Rightarrow H$, H being a conjunction of an increasingly large number of constraints generated from F "on demand": a constraint is added only if it is violated by the current candidate solution. This resembles quantifier elimination algorithms we have proposed for linear real arithmetic [17]; one difference is that the termination argument is simpler: with a finite number n of Booleans as free variables, a new added constraint suppresses at least one of the 2^n models, thus there can be at most 2^n iterations; in comparison the termination arguments for arithmetic involve counting projections of polyhedra.

Reductions from invariant inferences to quantifier elimination, or to minimization constrained by a universally quantified formula, have already been proposed for numerical constraints, where the unknowns are numerical quantities, in contrast to the present work where they are Booleans [16].

Reductions from loop invariant inference in predicate abstraction to Boolean constraint solving were introduced in [9], but that work assumed a postcondition to prove, as opposed to minimizing the result. The problem we solve is the same as the one from the later work [23, Sec. 5], but instead of concretely enumerating the (potentially exponential) set of paths inside the program (corresponding to all disjuncts in a disjunctive normal form of the transition relation), each path corresponding to one constraint, we lazily enumerate witnesses for such paths. Unfortunately, we do not have an implementation of the algorithm from [23] at our disposal for performance comparisons.

More generally, a number of approaches for invariant inference based on constraint solving have been proposed in the last years, especially for reducing numerical invariant inference to numerical constraint solving [8,4] or mathematical programming [6]. One difference between these constraint approaches and ours, except that our variable are Boolean and theirs are real, is that we use a *lazy* constraint generation scheme: we generate constraints only when a candidate solution violates them, a method long known in mathematical programming when applying *cuts*. We applied a similar technique for quantifier elimination for linear real arithmetic, using lazy conversions to conjunctive normal form [17]. A recent *max-policy iteration* considers each path through the loop as a constraint, and lazily selects a combination of paths, using SMT-solving to point the next relevant path [5].

6 Conclusion

We have given algorithms for finding loop invariants, or, equivalently, invariants for reactive nodes, given as templates with Boolean parameters. Using disjunctive invariants for reactive nodes, one obtains an abstraction of the reactive node as a finite automaton with transitions labeled with guards over node inputs.

If a system consists of a number of nodes, then some of these nodes may be replaced by their abstract automaton, resulting in a more abstract system whose behaviors include all behaviors of the original system. This new system can in turn be analyzed by the same method. Thus, our method supports modular and compositional analysis.

We provide the Candle tool, built using the Yices SMT-solver and the Mjollnir quantifier elimination procedure, which computes abstractions of Lustre nodes.

References

1. Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Refining the control structure of loops using static analysis. In: EMSOFT, pp. 49–58. ACM, New York (2009)
2. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: POPL (Symposium on Principles of programming languages), pp. 178–188. ACM (1987)
3. Cimatti, A.: Beyond Boolean SAT: Satisfiability modulo theories. In: Discrete Event Systems, WODES, pp. 68–73 (May 2008)
4. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: 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)
5. Gawlitza, T.M., Monniaux, D.: Improving Strategies via SMT Solving. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 236–255. Springer, Heidelberg (2011)
6. Goubault, E., Roux, S.L., Leconte, J., Liberti, L., Marinelli, F.: Static analysis by abstract interpretation: A mathematical programming approach. Electr. Notes Theor. Comput. Sci. 267(1), 73–87 (2010)
7. 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)
8. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 281–292. ACM, New York (2008)
9. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-Based Invariant Inference over Predicate Abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2009)
10. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: Zorn, B.G., Aiken, A. (eds.) PLDI, pp. 292–304. ACM (2010)
11. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Cimatti, R.B.A., Jones (eds.) Formal Methods in Computer-Aided Design (FMCAD), pp. 109–117. IEEE (2008)
12. Jeannet, B.: Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones. Ph.D. thesis, Institut National Polytechnique de Grenoble (September 2000)
13. Jeannet, B.: Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. Formal Methods in System Design 23, 5–37 (2003)
14. Jeannet, B., Halbwachs, N., Raymond, P.: Dynamic Partitioning in Analyses of Numerical Properties. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 39–50. Springer, Heidelberg (1999)
15. 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)
16. Monniaux, D.: Automatic modular abstractions for template numerical constraints. Logical Methods in Computer Science (June 2010)

17. Monniaux, D.: Quantifier Elimination by Lazy Model Enumeration. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 585–599. Springer, Heidelberg (2010)
18. Popeea, C., Chin, W.-N.: Inferring Disjunctive Postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)
19. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM TOPLAS 29 (August 2007)
20. Sankaranarayanan, S., Ivančić, 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)
21. Shankar, N.: Symbolic Analysis of Transition Systems. In: Gurevich, Y., Kutter, P.W., Vetta, A., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 287–302. Springer, Heidelberg (2000)
22. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying Loop Invariant Generation using Splitter Predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011)
23. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. SIGPLAN Not. 44, 223–234 (2009)

Template-Based Unbounded Time Verification of Affine Hybrid Automata*

Thao Dang[1,**] and Thomas Martin Gawlitza[1,2]

[1] Verimag
{Thao.Dang,Thomas.Gawlitza}@imag.fr
[2] University of Sydney
TGawlitza@usyd.edu.au

Abstract. Computing over-approximations of all possible time trajectories is an important task in the analysis of hybrid systems. Sankaranarayanan et al. [20] suggested to approximate the set of reachable states using template polyhedra. In the present paper, we use a max-strategy improvement algorithm for computing an abstract semantics for affine hybrid automata that is based on template polyhedra and safely over-approximates the concrete semantics. Based on our formulation, we show that the corresponding abstract reachability problem is in co−NP. Moreover, we obtain a polynomial-time algorithm for the time elapse operation over template polyhedra.

1 Introduction

Motivation. Hybrid systems have become widely accepted as a mathematical model appropriate for embedded systems and cyber-physical systems since they allow to describe the mixed discrete-continuous dynamics resulting from integrations of computations and physical processes. Verification is one of the most important questions in the design of such systems. For safety properties, this often leads to reachability analysis. The essential idea of many existing reachability computation techniques could be roughly described as tracking the evolution of the reachable set under the continuous flows using some set represention (such as polyhedra, ellipsoids, level sets, support functions)[1]. Since exact computation is possible only for restrictive classes of continuous dynamics, reachable sets are often approximated using time discretization. Such step-by-step tracking processes can be expensive when time steps should be small for accuracy reasons, and moreover discrete transitions can significantly increase the geometric complexity of reachable sets. This is a reason, besides undecidability of the reachability problem for general hybrid systems, why *unbounded time* reachability computation remains a challenge. Another category of techniques aim at finding approximations which might not be precise but good enough to prove a property of interest. Among such techniques, we can mention the works on barrier certificates [18],

* This work was partially funded by the ANR project VEDECY.

** VERIMAG is a joint laboratory of CNRS, Université Joseph Fourier and Grenoble INP.

[1] The hybrid systems reachability analysis literature is vast. The reader is referred to the recent proceedings of the conferences Hybrid Systems: Control and Computation.

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 34–49, 2011.

polynomial invariants [23] and polyhedral invariants [20]), and various discrete abstraction techniques [3–5, 22]). The work we present in this paper is close to the techniques of the second category, in particular to the work by Sankaranarayanan et al. [20].

Sankaranarayanan et al. [20] suggested to approximate the set of reachable states by template polyhedra. Their work is focused on studying the *time elapse operation* for affine hybrid automata over template polyhedra, since this is a challenging problem in hybrid systems verification. They in particular adapted the min-strategy iteration approach of Costan et al. [6] in order to compute a small template polyhedron that safely over-approximates the set of states reachable by continuous evolution. at a single location. Each min-strategy improvement step can be performed in polynomial time through linear programming. The approximation of the set of reachable states their algorithm computes can be used to improve an existing flowpipe construction technique using Taylor series [20]. However, their approach for performing the time elapse operation has disadvantages: (1) Their min-strategy iteration algorithm does not guarantee minimality of the computed template polyhedron. In fact, its accuracy heavily depends on the staying conditions (also called location invariants). If there are no restrictions due to staying conditions, then their algorithm will return too conservative approximations in many cases. (2) The number of min-stratgies is double exponential and a polynomial upper bound for the number of min-strategy improvement steps their algorithm performs is not known.

Contributions. In this paper we propose a remedy for the mentioned disadvantages of the approach of Sankaranarayanan et al. [20]. Moreover, instead of only focusing on the time elapse operation, we study the more general problem of computing *abstract semantics* for affine hybrid automata w.r.t. given linear templates — a problem which is useful for *unbounded time verification*. We emphasize that we provide a max-strategy improvement algorithm that *precisely* computes these abstract semantics and not just safely over-approximates it, as it is often done when using the widening/narrowing approach of Cousot and Cousot [7].

To this end, we firstly reduce our problem to the problem of computing least solutions of systems of inequalities of the form $\mathbf{x}_i \geq f(\mathbf{x}_1, \ldots, \mathbf{x}_n)$, where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are variables that take values from $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$ and f is an operator of a special structure that is in particular monotone and concave (cf. Gawlitza and Seidl [10, 11, 12, 14, 15]). Our max-strategy improvement algorithm for solving these systems of inequalities performs at most exponentially many strategy improvement steps, each of which can be performed in polynomial-time through linear programming. Although only an exponential upper bound is known, the hope is that only a few strategy improvement steps are required for typical examples. As a byproduct of our considerations, we show that the corresponding abstract reachability problem is in $\mathsf{co}-\mathsf{NP}$. When we apply our method to perform just the time elapse operation, our max-strategy improvement algorithm will perform at most polynomially many strategy improvement steps. Hence, we provide a polynomial-time algorithm for the time elapse operation for affine hybrid automata over template polyhedra.

Related Work. The concepts we present in this paper, strictly generalize the concepts studied by Gawlitza and Seidl [10]. This is no surprise, since affine hybrid automata

are a strict generalization of the affine programs considered by Gawlitza and Seidl [10]. The additional challenge comes from the time elapse operation. The approach of Gawlitza and Seidl [10] and the approach we present in this paper are both based on max-strategy iteration. Costan et al. [6] were the first who suggested to use strategy iteration for computing numerical invariants (for instance w.r.t. to template polyhedra). Strategy iteration can be seen as an alternative to the traditional widening/narrowing approach of Cousot and Cousot [7]. For more information regarding these approaches see Adjé et al. [1, 2], Costan et al. [6], Gaubert et al. [9], Gawlitza and Seidl [10, 11, 12], Gawlitza and Monniaux [13], Gawlitza and Seidl [14, 15].

Corresponding Technical Report. Omitted proofs and reports on our proof-of-concept implementation can be found in the corresponding technical report [8].

2 Basics

Notations. The set of real numbers is denoted by $\mathbb{R}$. The complete linearly ordered set $\mathbb{R} \cup \{-\infty, \infty\}$ is denoted by $\overline{\mathbb{R}}$. The transposed of a matrix A is denoted by $A^\top$. 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 $f : X \to Y^k$, i.e., $f_{i\cdot}(x) = (f(x))_{i\cdot}$ for all $x \in X$ and all $i \in \{1, \dots, k\}$. For $x, y \in \overline{\mathbb{R}}^n$, we write $x \leq y$ iff $x_{i\cdot} \leq y_{i\cdot}$ for all $i \in \{1, \dots, n\}$. $\overline{\mathbb{R}}^n$ is partially ordered by $\leq$. We write $x < y$ iff $x \leq y$ and $x \neq y$. The elements x and y are called *comparable* iff $x \leq y$ or $y \leq x$.

Let $\mathbb{D}$ be a partially ordered set. We denote the *least upper bound* and the *greatest lower bound* of a set $X \subseteq \mathbb{D}$ by $\bigvee X$ and $\bigwedge X$, respectively, provided that they exist. Their existence is in particular guaranteed if $\mathbb{D}$ is a *complete lattice*. The least element $\bigvee \emptyset$ (resp. the greatest element $\bigwedge \emptyset$) is denoted by $\bot$ (resp. $\top$), provided that it exists. We define the binary operators $\vee$ and $\wedge$ by $x \vee y := \bigvee\{x, y\}$ and $x \wedge y := \bigwedge\{x, y\}$ for all $x, y \in \mathbb{D}$, respectively. If $\mathbb{D}$ is a *linearly ordered set* (for instance $\mathbb{R}$ or $\overline{\mathbb{R}}$), then $\vee$ is the *maximum* operator and $\wedge$ the *minimum* operator.

A function $f : \mathbb{D}_1 \to \mathbb{D}_2$, where $\mathbb{D}_1$ and $\mathbb{D}_2$ are partially ordered sets, is called *monotone* iff $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in \mathbb{D}_1$. The fixpoint theorem of Knaster/Tarski [21] states that any monotone self-map $f : \mathbb{D} \to \mathbb{D}$ on a complete lattice $\mathbb{D}$ has a least fixpoint $\mu f = \bigwedge\{x \in \mathbb{D} \mid x \geq f(x)\}$.

A mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ is called *affine* iff there exist $A \in \mathbb{R}^{m \times n}$ and $b \in \overline{\mathbb{R}}^m$ such that $f(x) = Ax + b$ for all $x \in \overline{\mathbb{R}}^n$. Observe that f is monotone, if all entries of A are non-negative. Here, we use the convention $-\infty + \infty = -\infty$. A mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}$ is called *weak-affine* iff there exist $a \in \mathbb{R}^n$ and $b \in \overline{\mathbb{R}}$ such that $f(x) = a^\top x + b$ for all $x \in \overline{\mathbb{R}}^n$ with $f(x) \neq -\infty$. Accordingly, a mapping $f : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}^m$ is called *weak-affine* iff there exist weak-affine mappings $f_1, \dots, f_m : \overline{\mathbb{R}}^n \to \overline{\mathbb{R}}$ such that $f = (f_1, \dots, f_m)$. Every affine mapping is weak-affine, but not vice-versa. In the following we are in particular interested in mappings that are point-wise minimums of finitely many monotone weak-affine mappings.

Hybrid Automata. In this paper, we study affine hybrid automata. Here, the continuous consecution at each location l is given by an affine vector field V and a staying condition

I that is a convex polyhedron. A vector field V over $\mathbb{R}^n$ is just an operator on $\mathbb{R}^n$. Hence, it can be defined by $V(x) = Ax + b$ for all $x \in \mathbb{R}^n$, where $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$. A *staying conditions* I is simply a subset of $\mathbb{R}^n$. We say that a differentiable time trajectory $\tau : [0, \delta] \to \mathbb{R}^n$ ($\delta \in \mathbb{R}_{\geq 0}$) *evolves from* $\tau(0)$ *to* $\tau(\delta)$ *according to the vector field* V *over* $\mathbb{R}^n$ *while satisfying the staying condition* $I \subseteq \mathbb{R}^n$ iff (1) $\dot{\tau}(t) = V(\tau(t))$ for all $t \in [0, \delta)$, and (2) $\tau(t) \in I$ for all $t \in [0, \delta]$.

Example 1 (Sankaranarayanan et al. [20]). We consider the affine vector field $V : \mathbb{R}^2 \to \mathbb{R}^2$ that is defined by

$$V(x) = Ax + b \qquad \text{for all } x \in \mathbb{R}^2, \text{ where } A = \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix} \text{ and } b = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$$

and the staying condition $I = (-\infty, 2.5] \times \mathbb{R}$ that is a convex polyhedron. The polyhedron $P = \{x \in \mathbb{R}^2 \mid x_1. \leq 2.5,\ x_2. \leq 2.5, \text{ and } x_2. \leq x_1.\}$ is an invariant in the following sense: each differentiable trajectory that starts in P and evolves according to V while satisfying I stays in P. The situation is illustrated in Figure 1. □

A *hybrid automaton* $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$ consists of the following components:

- n is the number of *continuous variables*.
- $\mathbf{L}$ is a finite set of *locations*.
- $l_0 \in \mathbf{L}$ is the *initial location*.
- $\mathcal{T}$ is a finite set of *discrete transitions*. Each transition $(l_1, \Xi, l_2) \in \mathcal{T}$ consists of a move from the location $l_1 \in \mathbf{L}$ to the location $l_2 \in \mathbf{L}$, and an assertion $\Xi \subseteq (\mathbb{R}^n)^2$.
- $\Theta \subseteq \mathbb{R}^n$ is the set of possible *initial values* of the continuous variables at l_0.
- $\mathbf{D}$ is a mapping that maps each location $l \in \mathbf{L}$ to a vector field $\mathbf{D}(l) : \mathbb{R}^n \to \mathbb{R}^n$.
- $\mathbf{I}$ is a mapping that maps each location $l \in \mathbf{L}$ to a *staying condition* $\mathbf{I}(l) \subseteq \mathbb{R}^n$.

At each location $l \in \mathbf{L}$, the values of the continuous variables evolve according to $\mathbf{D}(l)$ while satisfying $\mathbf{I}(l)$. The assertion $\Xi \subseteq (\mathbb{R}^n)^2$ of a discrete transition $(l, \Xi, l') \in \mathcal{T}$ combines a guard with an assignment.

A hybrid automaton $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$ is called *affine* iff the following statements are fulfilled: (1) The initial condition, location invariants and transition relations are all convex polyhedra.[2] (2) The dynamics $\mathbf{D}(l)$ at each location $l \in \mathbf{L}$ is an affine vector field.

We now introduce our running example. We choose a simple example without discrete transitions, since the main challenges stem from the time elapse operation on which we want to focus in this paper.

Example 2. An affine hybrid automaton is given by $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0\}$, where $n = 2$, $\mathbf{L} = \{1\}$, $\mathcal{T} = \emptyset$, $\Theta = \{x \in \mathbb{R}^2 \mid x_1., x_2. \leq 1 \text{ and } x_2. \leq x_1.\}$, $\mathbf{D}(1) = V$, $\mathbf{I}(1) = I$, and $l_0 = 1$. V and I are defined in Example 1. □

A computation of a hybrid automaton is a possibly infinite sequence $(l_0, x_0), (l_1, x_1), \ldots$, where $x_0 \in \Theta$ and, for all $i \in \mathbb{N}$, one of the following statements hold: *(Discrete Consecution)* There exists a discrete transition $(l_i, \Xi, l_{i+1}) \in \mathcal{T}$ such that $(x_i, x_{i+1}) \in \Xi$. *(Continuous Consecution)* $l_i = l_{i+1}$ and there exists a $\delta \in \mathbb{R}_{>0}$ and a differentiable time trajectory $\tau : [0, \delta]$ that evolves from x_i to x_{i+1} according to the vector field $\mathbf{D}(l_i)$ while satisfying the staying condition $\mathbf{I}(l_i)$.

[2] Here, we identify $(\mathbb{R}^n)^2$ with $\mathbb{R}^{2n}$.

Template Polyhedra. As an abstract domain [7] we use template polyhedra as introduced by Sankaranarayanan et al. [19]. For that we fix a *template constraint matrix* $T \in \mathbb{R}^{m\times n}$, where we w.l.o.g. assume that $T_{i\cdot} \neq (0,\ldots,0)$ for every $i \in \{1,\ldots,m\}$. Each row of T represents a linear template. Each template relates n variables. The *concretization* $\gamma : \overline{\mathbb{R}}^m \to 2^{\mathbb{R}^n}$ and the *abstraction* $\alpha : 2^{\mathbb{R}^n} \to \overline{\mathbb{R}}^m$ are defined as follows:

$$\gamma(d) := \{x \in \mathbb{R}^n \mid Tx \leq d\}\, \forall d \in \overline{\mathbb{R}}^m,\ \alpha(X) := \bigwedge\{d \in \overline{\mathbb{R}}^m \mid \gamma(d) \supseteq X\}\, \forall X \subseteq \mathbb{R}^n,$$

As shown by Sankaranarayanan et al. [19], α and γ form a Galois connection, i.e., for all $X \subseteq \mathbb{R}^n$ and all $d \in \overline{\mathbb{R}}^m$, $\alpha(X) \leq d$ iff $X \subseteq \gamma(d)$. Hence, $\alpha \circ \gamma$ is a downwards closure operator, and $\gamma \circ \alpha$ is an upwards closure operator[3]. This in particular implies that $\alpha \circ \gamma$ and $\gamma \circ \alpha$ are monotone. In order to simplify notations, we denote $\alpha \circ \gamma$ by $\mathbf{cl}$. The abstract elements from $\alpha(2^{\mathbb{R}^n}) = \mathbf{cl}(\overline{\mathbb{R}}^m)$ are called *closed*. The convex polyhedra from the set $\gamma(\overline{\mathbb{R}}^m) = \gamma(\alpha(2^{\mathbb{R}^n}))$ are called *template polyhedra*.

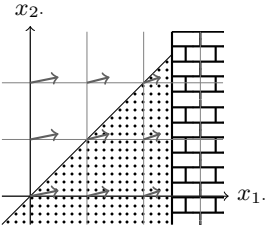


Fig. 1. Illustration for Example 1. The dotted region represents the convex polyhedron P. The wall represents the region that is not allowed, because of the staying condition I. The arrows illustrate the directions of the vector field V. Observe that any trajectory that starts in P and evolves according to V while satisfying I will stay in P.

Example 3. Let the template constraint matrix $T \in \mathbb{R}^{3\times 2}$ and $d \in \overline{\mathbb{R}}^3$ be defined by

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 1 \end{pmatrix}, \text{ and } \qquad d = \begin{pmatrix} 2.5 \\ 2.5 \\ 0 \end{pmatrix}.$$

Then $\gamma(d) = P$, where P is defined in Example 1 (see Figure 1). □

The following properties of the operator $\mathbf{cl}$ will be crucial for the algorithms we present in this paper:

Lemma 1. *For all $i \in \{1,\ldots,m\}$ and all $d \in \overline{\mathbb{R}}^m$, we have:*

1. $\mathbf{cl}_{i\cdot}(d) = \sup\{T_{i\cdot}x \mid x \in \mathbb{R}^n \text{ and } Tx \leq d\}$
2. $\mathbf{cl}$ *is a point-wise minimum of finitely many monotone weak-affine mappings.*

[3] An operator $f : \mathbb{D} \to \mathbb{D}$ on a partially ordered set $\mathbb{D}$ is called *downwards (resp. upwards) closure operator* iff (1) f is monotone, (2) f is idempotent (i.e. $f^2 = f$), and (3) $f(x) \subseteq x$ (resp. $f(x) \supseteq x$) for all $x \in \mathbb{D}$.

Proof. For the first statement see Sankaranarayanan et al. [19]. In order to show that $\mathbf{cl}$ is a point-wise minimum of finitely many monotone weak-affine mappings, we use the strong duality theorem for linear programming as follows: $\mathbf{cl}_{i\cdot}(d) = \sup\{T_{i\cdot}x \mid x \in \mathbb{R}^n \text{ and } Tx \leq d\} = \inf\{d^\top y \mid y \in \mathbb{R}^m_{\geq 0},\ T^\top y = T_{i\cdot}^\top\}$ for all d with $\gamma(d) \neq \emptyset$. This gives us the statement. □

Invariants and Positive Invariants. Let $V : \mathbb{R}^n \to \mathbb{R}^n$ be a vector field and $I \subseteq \mathbb{R}^n$ a staying condition. A set $X \subseteq \mathbb{R}^n$ is called an *invariant* of (V, I) iff every trajectory that starts in X and evolves according to V while satisfying I stays in X. Before going further, we introduce the following notation: For all $d \in \overline{\mathbb{R}}^m$ and all $R \subseteq \{1, \ldots, m\}$, we define $d|_R \in \overline{\mathbb{R}}^m$ by

$$(d|_R)_{i\cdot} = \begin{cases} d_{i\cdot} & \text{if } i \in R \\ \infty & \text{if } i \notin R \end{cases} \qquad \text{for all } i \in \{1, \ldots, m\}.$$

Assume now that the affine vector field V is affine, and the staying condition I is a template polyhedron, i.e., $I \in \gamma(\overline{\mathbb{R}}^m)$. A template polyhedron $P \in \gamma(\overline{\mathbb{R}}^m)$ is called a *positive invariant* of (V, I) iff there exists some $R \subseteq \{1, \ldots, m\}$ such that the following statements hold:

1. $T_{i\cdot}V(x) \leq 0$ for all $x \in P$ with $T_{i\cdot}x = \alpha_{i\cdot}(P)$ and all $i \in R$ with $\alpha_{i\cdot}(P) < \alpha_{i\cdot}(I)$.
2. $P = \gamma(\alpha(P)|_R)$.

Our notion of positive invariants slightly differs from the notion of positive invariants of Sankaranarayanan et al. [20]. However, observe that every positive invariant in the sense of Sankaranarayanan et al. [20] is a positive invariant in our sense.

Every positive invariant is an invariant. However, there exist template polyhedra that are invariants without being positive invariants. Indeed due to the presence of staying conditions, for a template polyhedron P to be an invariant, the above condition $T_{i\cdot}V(x) \leq 0$ does not need to be satisfied at all the points $x \in P$ on the face corresponding to $T_{i\cdot}x = \alpha_{i\cdot}(P)$, when $\alpha_{i\cdot}(P) < \alpha_{i\cdot}(I)$. However, because of lack of space and additionally for clarity of presentation, we do not consider this in the present paper.

Example 4. We continue our running example (see Figure 1), i.e., the affine vector field V and the staying condition I are defined in Example 1, and the template constraint matrix T is defined in Example 3. The staying condition I is a template polyhedron, since $I = \gamma((2.5, \infty, \infty)^\top)$. The template polyhedron $P = \gamma(d)$ is an invariant as well as a positive invariant of (V, I). If we choose $R = \{1, 3\}$, then the requirements of the definition can be verified easily (cf. Figure 1). □

Our Goals: Time Elapse Operations and Abstract Semantics. We are interested in computing abstract semantics for affine hybrid automata w.r.t. template polyhedra. Performing the time elapse operation w.r.t. to template polyhedra is just the special case, where the affine hybrid automaton has no discrete transitions.

The *abstract semantics* for the affine hybrid automaton $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$ (w.r.t. the template polyhedra domain) is the point-wise minimal mapping $V^\sharp_\square$ that maps

every location $l \in \mathbf{L}$ to a template polyhedron $V^\sharp_\square[l] \in \gamma(\overline{\mathbb{R}}^m)$ and fulfills the following constraints: (1) $V^\sharp_\square[l_0] \supseteq \Theta$. (2) $V^\sharp_\square[l]$ is a positive invariant of $(\mathbf{D}(l), \mathbf{I}(l))$ for every location $l \in \mathbf{L}$. (3) $x' \in V^\sharp_\square[l']$ for all discrete transitions $(l, \Xi, l') \in \mathcal{T}$ and all $(x, x') \in \Xi$ with $x \in V^\sharp_\square[l]$. The existence of such a point-wise minimal mapping will be ensured by our findings. The abstract semantics safely over-approximates the concrete semantics.

In order to verify safety properties, a problem one is interested in is *abstract reachability*, which is the following decision problem: Decide whether or not, for a given template constraint matrix $T \in \mathbb{R}^{m \times n}$, a given affine hybrid automaton $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$, and a given location $l \in L$, the statement $V^\sharp[l] \neq \emptyset$ holds.

In this paper, we will adapt the max-strategy improvement algorithm of Gawlitza and Seidl [10, 11, 12, 14, 15] for computing $V^\sharp_\square$. We will find that abstract reachability is in $\mathsf{co{-}NP}$. Whether or not it is also in P is an open question. However, we at least know that it is a hard problem in the following sense: a polynomial-time algorithm for abstract reachability would give us a polynomial-time algorithm for computing the winning regions of mean-payoff games (see [12]). The latter problem is in $\mathsf{UP} \cap \mathsf{co{-}UP}$ (see Jurdzinski [16]) and it is a long outstanding and fundamental question whether or not it is in P.

The problem of performing the time elapse operation over template polyhedra is the following computational problem: Compute, for a given template constraint matrix T, a given affine vector field $V : \mathbb{R}^n \to \mathbb{R}^n$ and given template polyhedra Θ and I with $\Theta \subseteq I$, the least positive invariant of (V, I) which is a superset of Θ. We will show that the latter computational problem can be solved in polynomial time.

3 Our Approach: Getting into the Corset of the Monotone Framework

We aim at adapting the max-strategy improvement algorithms of Gawlitza and Seidl [10, 11, 12, 14, 15] in order to obtain an algorithm for computing abstract semantics. For that we have to formulate the problem as a problem of finding the least fixpoint of a self-map that is a maximum of finitely many monotone and concave self-maps (cf. Gawlitza and Seidl [10, 14]). The challenge is to get the time elapse operation into the corset of this monotone framework.

The Time Elapse Operation. Let $V : \mathbb{R}^n \to \mathbb{R}^n$ be an affine vector field. Firstly, we define the operator Δ^V on $\overline{\mathbb{R}}^m$ by

$$\Delta^V_{k.}(d) := \sup \{T_{k.}V(x) \mid x \in \mathbb{R}^n,\ Tx \leq d,\ T_{k.}x \geq d_{k.}\}$$

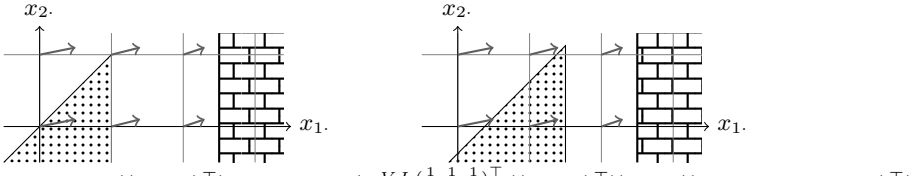
for all $k \in \{1, \ldots, m\}$ and all $d \in \overline{\mathbb{R}}^m$ with $d_{k.} < \infty$. Note that $\Delta^V_{k.}(d) = -\infty$, whenever $\{x \in \mathbb{R}^n \mid Tx \leq d, T_{k.}x \geq d_{k.}\} = \emptyset$. This is in particular fulfilled, if there exists some $i \in \{1, \ldots, m\}$ with $d_{i.} = -\infty$. Moreover, we set $\Delta^V_{k.}(d) := 0$ for all $k \in \{1, \ldots, m\}$ and $d \in \overline{\mathbb{R}}^m$ with $d_{k.} = \infty$. Intuitively, $\Delta^V_{k.}(d) > 0$ iff there exists some point x on the face $\mathcal{F} := \{x \in \mathbb{R}^n \mid Tx \leq d, T_{k.}x \geq d_{k.}\}$ such that $V(x)$ points to the outside.

For all $\epsilon \in \mathbb{R}^m_{>0}$, we define the operator $f^{V,\epsilon}$ on $\overline{\mathbb{R}}^m$ by

$$f^{V,\epsilon}(d) := d + \epsilon^\top \Delta^V(d) \qquad \text{for all } d \in \overline{\mathbb{R}}^m.$$

An application of the operator $f^{V,\epsilon}$ corrects the bounds to the templates according to the vector field V, ignoring the staying condition I. In order to take the staying condition I into account, we assume w.l.o.g. that I is a template polyhedron, i.e., $I \in \gamma(\overline{\mathbb{R}}^m)$. For all $\epsilon \in \mathbb{R}^m_{>0}$, we define the operator $F^{V,I,\epsilon}$ on $\overline{\mathbb{R}}^m$ as follows:

$$F^{V,I,\epsilon}(d) := f^{V,\epsilon}(d) \wedge \alpha(I) \qquad \text{for all } d \in \overline{\mathbb{R}}^m$$



(a) $\gamma((1,1,0)^\top)$ (b) $\gamma(F^{V,I,(\frac{1}{8},\frac{1}{8},\frac{1}{8})^\top}((1,1,0)^\top)) = \gamma((1.5, 1.125, -0.375)^\top)$

Fig. 2. The Running Example: An Application of $F^{V,I,\epsilon}$ for $\epsilon = (\frac{1}{8}, \frac{1}{8}, \frac{1}{8})^\top$

How the operator $F^{V,I,\epsilon}$ modifies a template polyhedron is shown in Figure 2 for our running example. Positive invariants can now be characterized as follows:

Lemma 2. *Let $\epsilon \in \mathbb{R}^m_{>0}$. For all $d \in \overline{\mathbb{R}}^m$ the following holds: The template polyhedron $\gamma(d)$ is a positive invariant of (V, I) iff $d \geq \mathbf{cl}(\mathbf{cl}(d) \vee F^{V,I,\epsilon}(\mathbf{cl}(d)))$.* □

In order to use the above lemma within a monotone framework, we have to ensure that $F^{V,I,\epsilon} \circ \mathbf{cl}$ is monotone. Then $\mathcal{F} := \mathbf{cl} \circ (\mathbf{cl} \vee F^{V,I,\epsilon} \circ \mathbf{cl})$ is monotone, too, and the fixpoint theorem of Knaster/Tarski [21] can be applied.[4] Observe that by construction $F^{V,I,\epsilon} \circ \mathbf{cl}$ is monotone, whenever $f^{V,\epsilon} \circ \mathbf{cl}$ is monotone. The operator $f^{V,\epsilon} \circ \mathbf{cl}$ is monotone on $\overline{\mathbb{R}}^m$, whenever the operator $f^{V,\epsilon}$ is monotone on $\mathbf{cl}(\overline{\mathbb{R}}^m)$.[5] If we choose ϵ small enough, then we enforce the monotonicity of $f^{V,\epsilon}$ on $\mathbf{cl}(\overline{\mathbb{R}}^m)$ and thus finally the monotonicity of $F^{V,I,\epsilon} \circ \mathbf{cl}$ and $\mathcal{F}$:

Lemma 3 (Monotonicity of $f^{V,\epsilon}$). *Assume $V(x) = Ax + b$ for all $x \in \mathbb{R}^n$. From A and T, we can compute an $\epsilon^{(0)} \in \mathbb{R}^m_{>0}$ in polynomial time such that $f^{V,\epsilon}$ is monotone on $\mathbf{cl}(\overline{\mathbb{R}}^m)$, whenever $\epsilon \leq \epsilon^{(0)}$. For every $\epsilon \leq \epsilon^{(0)}$, $f^{V,\epsilon} \circ \mathbf{cl}$ is a point-wise minimum of finitely many monotone weak-affine self-maps.* □

Because of the above lemma, we from now on assume that we have chosen an $\epsilon \in \mathbb{R}^m_{>0}$ such that $f^{V,\epsilon} \circ \mathbf{cl}$ and thus finally $\mathbf{cl} \circ (\mathbf{cl} \vee F^{V,I,\epsilon} \circ \mathbf{cl}) = \mathbf{cl} \circ (\mathrm{id} \vee F^{V,I,\epsilon}) \circ \mathbf{cl}$ is monotone. Therefore, for all sets $\Theta \subseteq \mathbb{R}^n$ of values, there exists a least positive invariant P of (V, I) which is a superset of Θ. It is given by

$$\gamma(\mu(\alpha(\Theta) \vee \mathbf{cl} \circ (\mathbf{cl} \vee F^{V,I,\epsilon} \circ \mathbf{cl})))$$

[4] For mappings $f, g : X \to \mathbb{D}$, $f \vee g$ denotes the mapping that is defined by $(f \vee g)(x) := f(x) \vee g(x)$ for all $x \in X$.

[5] It is not always possible to choose an ϵ such that $f^{V,\epsilon}$ is monotone on $\overline{\mathbb{R}}^m$.

However, we do not use this formulation. We want to have a simpler formulation that will allow us to perform the time elapse operation in polynomial time. For that we use the following fixpoint transfer lemma:

Lemma 4. *Let $\epsilon \in \mathbb{R}^m_{>0}$ be chosen such that $F^{V,I,\epsilon} \circ \mathbf{cl}$ is monotone. For all closed $\theta \in \mathbf{cl}(\overline{\mathbb{R}}^m)$, we have $\gamma(\mu(\theta \vee \mathbf{cl} \circ (\mathbf{cl} \vee F^{V,I,\epsilon} \circ \mathbf{cl}))) = \gamma(\mu(\theta \vee F^{V,I,\epsilon} \circ \mathbf{cl}))$.*[6] □

Putting everything together, we obtain our main result for the time elapse operation:

Theorem 1 (The Time Elapse Operation). *Let $V : \mathbb{R}^n \to \mathbb{R}^n$ be an affine vector field, and $\Theta, I \in \gamma(\overline{\mathbb{R}}^m)$ template polyhedra. Assume that $\epsilon \in \mathbb{R}^m_{>0}$ is chosen such that $F^{V,I,\epsilon} \circ \mathbf{cl}$ is monotone. The template polyhedron $\gamma(\mu(\alpha(\Theta) \vee F^{V,I,\epsilon} \circ \mathbf{cl}))$ is the least positive invariant of (V, I) which is a superset of Θ.*

Proof. The existence of ϵ is ensured by Lemma 3. The existence of the least fixpoint is ensured by the fixpoint theorem of Knaster/Tarski. Lemmata 2 gives us that $P := \gamma(\mu(\alpha(\Theta) \vee \mathbf{cl} \circ (\mathbf{cl} \vee F^{V,I,\epsilon} \circ \mathbf{cl})))$ is the least positive invariant of (V, I) which is a superset of Θ. Lemma 4 finally gives us $P = \gamma(\mu(\alpha(\Theta) \vee F^{V,I,\epsilon} \circ \mathbf{cl}))$. □

Abstract Semantics. So far, we have ignored the discrete transitions. In order to take them into account, we define an abstract semantics for discrete transitions $(l, \Xi, l') \in \mathcal{T}$. Recall that the assertion $\Xi \subseteq \mathbb{R}^{2n}$ is a convex polyhedron. In the following we will always assume that the convex polyhedron Ξ is given by a matrix $A_\Xi \in \mathbb{R}^{l \times 2n}$ and a vector $b_\Xi \in \mathbb{R}^l$ such that $\Xi = \{x \in \mathbb{R}^{2n} \mid A_\Xi x \leq b_\Xi\}$. The collecting semantics $[\![\Xi]\!]$ of Ξ is defined by $[\![\Xi]\!](X) := \{y \in \mathbb{R}^n \mid \exists x \in X \,.\, (x, y) \in \Xi\}$ for all $X \subseteq \mathbb{R}^n$. The abstract semantics $[\![\Xi]\!]^\sharp$ of Ξ is defined by $[\![\Xi]\!]^\sharp := \alpha \circ [\![\Xi]\!] \circ \gamma$. The abstract semantics safely over-approximates the collecting semantics and the concrete semantics. For all $k \in \{1, \ldots, m\}$ and all $d \in \overline{\mathbb{R}}^m$, we have:

$$[\![\Xi]\!]^\sharp_k(d) := \sup \left\{ T_k.y \;\middle|\; x \in \gamma(d),\ \begin{pmatrix} x \\ y \end{pmatrix} \in \Xi \right\} \tag{1}$$

$$= \sup \left\{ T_k.y \;\middle|\; \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^{2n},\ Tx \leq d,\ A_\Xi \begin{pmatrix} x \\ y \end{pmatrix} \leq b_\Xi \right\} \tag{2}$$

If we consider the dual of the above linear programming problem, we get that also the operator $[\![\Xi]\!]^\sharp$ on $\overline{\mathbb{R}}^m$ has nice properties (cf. Gawlitza and Seidl [10, 14]):

Lemma 5 (The Abstract Semantics $[\![\Xi]\!]^\sharp$). *The following holds for every convex polyhedron $\Xi \subseteq \mathbb{R}^{2n}$: (1) $[\![\Xi]\!]^\sharp = [\![\Xi]\!]^\sharp \circ \mathbf{cl} = \mathbf{cl} \circ [\![\Xi]\!]^\sharp$. (2) $[\![\Xi]\!]^\sharp$ is the point-wise minimum of finitely many monotone weak-affine operators.* □

We are now going to define an abstract semantics $V^\sharp$ for an *affine* hybrid automata $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$ that corresponds to the abstract semantics $V^\sharp_\square$ of Ψ (cf. Section 2). W.o.l.g. we assume that the initial condition Θ, the location invariants $\mathbf{I}(l)$, $l \in \mathbf{L}$, and the transition relations are all template polyhedra. The abstract semantics $V^\sharp$ of Ψ is the least solution of the following constraint system:

$$\mathbf{V}^\sharp[l_0] \geq \alpha(\Theta) \tag{3}$$

[6] Here, θ denotes the function that returns θ for every argument.

$$\mathbf{V}^\sharp[l] \geq F^{\mathbf{D}(l),\mathbf{I}(l),\epsilon(l)}(\mathbf{cl}(\mathbf{V}^\sharp[l])) \qquad \text{for all } l \in \mathbf{L} \tag{4}$$

$$\mathbf{V}^\sharp[l'] \geq [\![\Xi]\!]^\sharp(\mathbf{V}^\sharp[l]) \qquad \text{for all } (l,\Xi,l') \in \mathcal{T} \tag{5}$$

The variables $\mathbf{V}^\sharp[l]$, $l \in \mathbf{L}$ take values from $\overline{\mathbb{R}}^m$. The existence of the least solution is ensured by the fixpoint theorem of Knaster/Tarski, since we assume that, for all locations $l \in \mathbf{L}$, $\epsilon(l) \in \mathbb{R}^m_{>0}$ is chosen such that $F^{\mathbf{D}(l),\mathbf{I}(l),\epsilon(l)}$ is monotone. The existence of such an $\epsilon(l)$ is again ensured by Lemma 3.

Constraint (3) takes all possible initial values of the continuous variables at the initial location l_0 into account. Constraint (4) ensures that the template polyhedron $\gamma(V^\sharp[l])$ is a positive invariant of $(\mathbf{D}(l),\mathbf{I}(l))$ (cf. Lemma 2). Constraint (5) ensures that the template polyhedron $\gamma(V^\sharp[l'])$ contains at least all values that can come through the discrete transition (l,Ξ,l'). By construction, we have:

Theorem 2. *$V^\sharp_\square[l] = \gamma(V^\sharp[l])$ for all locations $l \in \mathbf{L}$.* □

Because of the above theorem, we should now aim at computing $V^\sharp$.

4 Adapting the Max-Strategy Approach

Notations. In this section, we consider systems $\mathcal{C}$ of inequalities of the form $\mathbf{x} \geq e$ (resp. $\mathbf{x} \leq e$), where $\mathbf{x}$ is a variable that takes values form $\overline{\mathbb{R}}$ and e is an expression over $\overline{\mathbb{R}}$. The set of variables of $\mathcal{C}$ is denoted by $\mathbf{X}_\mathcal{C}$, where we omit the subscript, whenever it is clear from the context. The semantics $[\![e]\!] : (\mathbf{X} \to \overline{\mathbb{R}}) \to \overline{\mathbb{R}}$ of an expression e is defined by $[\![\mathbf{x}]\!](\rho) := \rho(\mathbf{x})$ 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 on $\overline{\mathbb{R}}$, $e_1,\ldots,e_k$ are expressions, and $\rho : \mathbf{X} \to \overline{\mathbb{R}}$ is a variable assignment.

For a system $\mathcal{C}$ of constraints of the form $\mathbf{x} \geq e$ (resp. $\mathbf{x} \leq e$), we define the operator $[\![\mathcal{C}]\!] : (\mathbf{X} \to \overline{\mathbb{R}}) \to \mathbf{X} \to \overline{\mathbb{R}}$ by

$$[\![\mathcal{C}]\!](\rho)(\mathbf{x}) := \bigvee\{[\![e]\!]\rho \mid \mathbf{x} \geq e \text{ belongs to } \mathcal{C}\}$$

(resp. $[\![\mathcal{C}]\!](\rho)(\mathbf{x}) := \bigwedge\{[\![e]\!]\rho \mid \mathbf{x} \geq e$ belongs to $\mathcal{C}\}$) for all variable assignments $\rho : \mathbf{X} \to \overline{\mathbb{R}}$ and all variables $\mathbf{x} \in \mathbf{X}$. Hence, ρ is a solution of $\mathcal{C}$ iff $\rho \geq [\![\mathcal{C}]\!](\rho)$ (resp. $\rho \leq [\![\mathcal{C}]\!](\rho)$). The least (resp. the greatest) solution of $\mathcal{C}$ is $\mu[\![\mathcal{C}]\!]$ (resp. $\nu[\![\mathcal{C}]\!]$). For a system $\mathcal{C}$ of inequalities of the form $\mathbf{x} \geq e$ and a pre-fixpoint ρ of the operator $[\![\mathcal{C}]\!]$ (i.e., $\rho \leq [\![\mathcal{C}]\!](\rho)$), $\mu_{\geq\rho}[\![\mathcal{C}]\!]$ denotes the least solution of $\mathcal{C}$ that is greater than or equal to ρ.

Rewriting the Abstract Semantic In-Equations. We now rewrite the abstract semantic in-equations (3) - (5) into a system $\mathcal{C}(\Psi)$ of in-equations of the form $\mathbf{x} \geq f(\mathbf{x}_1,\ldots,\mathbf{x}_k)$, where the variables take values from $\overline{\mathbb{R}}$ and the operator f is a maximum of finitely many monotone weak-affine operators. The set $\mathbf{X}$ of variables of the system $\mathcal{C}(\Psi)$ of in-equations we are going to construct is $\mathbf{X} = \{\mathbf{d}_{l,i} \mid l \in \mathbf{L} \text{ and } i \in \{1,\ldots,m\}\}$. Corresponding to constraint (3) we add the following in-equations:

$$\mathbf{d}_{l_0,i} \geq \alpha_{i\cdot}(\Theta) \qquad \text{for all } i \in \{1,\ldots,m\} \tag{6}$$

Corresponding to constraint (4), for every location $l \in \mathbf{L}$, we add the following in-equations that will deal with the time elapse operation:

$$\mathbf{d}_{l,i} \geq F_{i\cdot}^{\mathbf{D}(l),\mathbf{I}(l),\epsilon(l)}(\mathbf{cl}((\mathbf{d}_{l,1},\ldots,\mathbf{d}_{l,m})^\top)) \qquad \text{for all } i \in \{1,\ldots,m\} \tag{7}$$

Corresponding to constraint (5), for every discrete transition $(l,\Xi,l') \in \mathcal{T}$, we add the following in-equations that will deal with the discrete transition (l,Ξ,l'):

$$\mathbf{d}_{l',i} \geq [\![\Xi]\!]^\sharp_{i\cdot}\left((\mathbf{d}_{l,1},\ldots,\mathbf{d}_{l,m})^\top\right) \qquad \text{for all } i \in \{1,\ldots,m\} \tag{8}$$

By construction we have:

Lemma 6. *Let $\rho^* : \mathbf{X} \to \overline{\mathbb{R}}$ be the least solution of $\mathcal{C}(\Psi)$. Then, for all locations $l \in \mathbf{L}$ and all $i \in \{1,\ldots,m\}$, we have $(V^\sharp[l])_{i\cdot} = \rho^*(\mathbf{d}_{l,i})$.* □

Example 5. We continue our running example, i.e., we aim at computing the abstract semantics $V^\sharp$ of the hybrid automaton Ψ from Example 2, where we use the template constraint matrix T introduced in Example 3. For that we choose $\epsilon(1) = (1,\ldots,1)^\top$. Then $f^{\mathbf{D}(1),\epsilon(1)}$ and thus $F^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}$ are monotone. The system $\mathcal{C}(\Psi)$ consists of the following in-equations:

$$\begin{aligned}
\mathbf{d}_{1,1} &\geq 1 & \mathbf{d}_{1,1} &\geq F_{1\cdot}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1},\mathbf{d}_{1,2},\mathbf{d}_{1,3})^\top))\\
\mathbf{d}_{1,2} &\geq 1 & \mathbf{d}_{1,2} &\geq F_{2\cdot}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1},\mathbf{d}_{1,2},\mathbf{d}_{1,3})^\top))\\
\mathbf{d}_{1,3} &\geq 0 & \mathbf{d}_{1,3} &\geq F_{3\cdot}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1},\mathbf{d}_{1,2},\mathbf{d}_{1,3})^\top))
\end{aligned}$$

Example 6 shows how we can compute the least solution of this constraint system. □

The Max-Strategy Improvement Algorithm. Let $\mathcal{C}$ be a system of inequalities of the form $\mathbf{x} \geq e$, where $[\![e]\!]$ is a point-wise minimum of finitely many monotone weak-affine operators. We aim at computing the least solution $\mu[\![\mathcal{C}]\!]$ of $\mathcal{C}$ that exists due to monotonicity.

A subset σ of $\mathcal{C}$ is called a max-strategy of $\mathcal{C}$ iff it contains exactly one constraint $\mathbf{x} \geq e$ for every variable $\mathbf{x}$ occurring in $\mathcal{C}$. For simplicity, we assume that in $\mathcal{C}$ there exists a constraint $\mathbf{x} \geq -\infty$ for every variable $\mathbf{x}$ occurring in $\mathcal{C}$. Then $\{\mathbf{x} \geq -\infty \mid \mathbf{x} \in \mathbf{X}\}$ is a max-strategy. This will be the max-strategy the algorithm starts with.

The max-strategy improvement algorithm maintains a current max-strategy σ and a current approximate $\rho : \mathbf{X} \to \overline{\mathbb{R}}$ to the least solution $\mu[\![\mathcal{C}]\!]$ of $\mathcal{C}$. The max-strategy algorithm can be written as follows:

Algorithm 1. The Max-Strategy Improvement Algorithm

$\sigma \leftarrow \{\mathbf{x} \geq -\infty \mid \mathbf{x} \in \mathbf{X}\}$; $\rho \leftarrow \{\mathbf{x} \mapsto -\infty \mid \mathbf{x} \in \mathbf{X}\}$;

while (ρ is not a solution of $\mathcal{C}$) {$\sigma \leftarrow$ improvement of σ w.r.t. ρ; $\rho \leftarrow \mu_{\geq\rho}[\![\sigma]\!]$; }

return ρ;

We have to define the term *improvement*. Let σ be a max-strategy of $\mathcal{C}$ and ρ be a pre-solution of $[\![\sigma]\!]$, i.e., $\rho \leq [\![\sigma]\!]\rho$. A max-strategy σ' of $\mathcal{C}$ is called an *improvement of σ w.r.t. ρ* iff the following conditions hold:

1. $[\![\sigma']\!]\rho \geq [\![\sigma]\!]\rho$.
2. If $\mathbf{x} \geq e$ belongs to σ and $\mathbf{x} \geq e'$ belongs to σ' with $e \neq e'$, then $[\![e']\!]\rho > [\![e]\!]\rho$.

The second condition ensured that a max-strategy is only changed at variables where we have a strict improvement. This is important for the correctness of the algorithm (cf. Gawlitza and Seidl [10, 11, 12, 14, 15]).

It is obvious that the algorithm returns the least solution of $\mathcal{C}$, whenever it terminates. From the considerations in the next subsection, it will follow that it terminates at the latest after considering every max-strategy at most once. In the next subsection, we will also explain how we can compute $\mu_{\geq\rho}[\![\sigma]\!]$ for a max-strategy σ and a variable assignment ρ that occurs during the run of the algorithm. Before doing so, we will use our algorithm for computing the abstract semantics of our running example.

Example 6. We apply the max-strategy improvement algorithm to the system $\mathcal{C}$ of constraints defined in Example 5. After the first max-strategy improvement step we may get the max-strategy σ_1 that consists of the following constraints:

$$\mathbf{d}_{1,1} \geq 1 \qquad \mathbf{d}_{1,2} \geq 1 \qquad \mathbf{d}_{1,3} \geq 0$$

We have to find the least solution ρ_1 of σ_1 that is greater than of equal to $\rho_0 = \{\mathbf{x} \mapsto -\infty \mid \mathbf{x} \in \mathbf{X}\}$. Hence, obviously $\rho_1 = \mu_{\geq\rho_0}[\![\sigma_1]\!] = \{\mathbf{d}_{1,1} \mapsto 1, \mathbf{d}_{1,2} \mapsto 1, \mathbf{d}_{1,3} \mapsto 0\}$. However, ρ_1 is not a solution of $\mathcal{C}$. Hence, we can improve the current max-strategy σ_1 w.r.t. ρ_1. We may obtain the max-strategy σ_2 that consists of the following constraints:

$$\mathbf{d}_{1,1} \geq F_{1.}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1}, \mathbf{d}_{1,2}, \mathbf{d}_{1,3})^\top))$$
$$\mathbf{d}_{1,2} \geq F_{2.}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1}, \mathbf{d}_{1,2}, \mathbf{d}_{1,3})^\top)) \qquad \mathbf{d}_{1,3} \geq 0$$

We get $\rho_2 = \mu_{\geq\rho_1}[\![\sigma_1]\!] = \{\mathbf{d}_{1,1} \mapsto 2.5, \mathbf{d}_{1,2} \mapsto 3.5, \mathbf{d}_{1,3} \mapsto 0\}$. How we can compute ρ_2 will be explained in Example 7. ρ_2 solves the constraint system $\mathcal{C}$. Hence, the algorithm terminates and returns ρ_2, which is the correct least solution of $\mathcal{C}$. Thus, we have $V^\sharp[1] = (2.5, 3.5, 0)^\top$. By Theorem 2, we get $V_\square^\sharp[1] = \gamma((2.5, 3.5, 0)^\top) = \{(x_1, x_2)^\top \in \mathbb{R}^2 \mid x_1 \leq 2.5, x_2 \leq 2.5, x_2 \leq x_1\}$ (cf. Figure 1). □

In the above example, we have 3 inequality constraints for each variable after introducing the constraints $\mathbf{d}_{1,i} \geq -\infty$, $i = \{1, 2, 3\}$. Hence, we have $3^3 = 9$ max-strategies. However, since the sequence of approximates is strictly increasing until it stabilizes, the constraint $\mathbf{d}_{1,1} \geq -\infty$ will not be considered after considering the constraint $\mathbf{d}_{1,1} \geq 1$. Similar, the constraint $\mathbf{d}_{1,1} \geq 1$ will not be considered after considering the constraint $\mathbf{d}_{1,1} \geq F_{1.}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1}, \mathbf{d}_{1,2}, \mathbf{d}_{1,3})^\top))$. Hence, the maximal number of max-strategies considered by our max-strategy improvement algorithm is $1 + 2 \cdot 3 = 7$.

This is not by accident. Whenever the affine hybrid system has exactly one location and no discrete transitions, the number of max-strategies the algorithm considers is at most $1 + 2m$. If we start the algorithm with the max-strategy that corresponds to the set Θ of all possible initial values, then we can reduce this number to $1 + m$. Thus, we have:

Lemma 7. *If we apply our max-strategy improvement algorithm for performing the time elapse operation, then the number of max-strategy improvement steps is bounded by m, where m is number of templates.* □

Evaluating a Single Max-Strategy. Let σ be a max-strategy for $\mathcal{C}(\Psi)$ and ρ be a variable assignment that occurs during a run of the max-strategy improvement algorithm (the constraint system $\mathcal{C}(\Psi)$ is defined in Subsection 4). We are aiming at computing $\mu_{\geq\rho}[\![\sigma]\!]$. For that, we firstly remove all constraints $\mathbf{x} \geq -\infty$ from σ and replace the corresponding variables with the constant $-\infty$. For simplicity, we denote the resulting system again by σ. Since the algorithm only improves max-strategies at positions where there are strict improvements, we have $\mu_{\geq\rho}[\![\sigma]\!](\mathbf{x}) > -\infty$ for all variables $\mathbf{x} \in \mathbf{X}$.

From the results of Gawlitza and Seidl [10, 14] it follows that $\mu_{\geq\rho}[\![\sigma]\!]$ equals the variable assignment $\rho^\sigma : \mathbf{X} \to \overline{\mathbb{R}}$ which is defined as follows:

$$\rho^\sigma(\mathbf{z}) := \sup\{\rho(\mathbf{z}) \mid \rho : \mathbf{X} \to \mathbb{R},\ \rho(\mathbf{x}) \leq [\![e]\!](\rho) \text{ for all constraints } \mathbf{x} \geq e \text{ of } \sigma\} \quad (9)$$

for all $\mathbf{z} \in \mathbf{X}$. Observe that the variable assignment $\rho^\sigma = \mu_{\geq\rho}(\sigma)$ only depends on the max-strategy σ and not on the variable assignment ρ. This is an important observation. Since the max-strategy improvement algorithm generates a strictly increasing sequence of variable assignments, each of which only depends on the current max-strategy, it follows that the max-strategy algorithm terminates at the latest after considering each max-strategy at most once.

In order to compute ρ^σ, we set up the system σ' of linear inequalities as follows: We start with an empty system of linear inequalities. For every inequality of the form $\mathbf{d} \geq c$, where $c \in \mathbb{R} \cup \{\infty\}$, we add the linear constraint $\mathbf{d} \leq c$. For every bunch of inequalities of the form

$$\mathbf{d}_1 \geq F^{V,I,\epsilon}_{1\cdot}(\mathbf{cl}((\mathbf{d}_1,\dots,\mathbf{d}_m)^\top)) \quad \cdots \quad \mathbf{d}_m \geq F^{V,I,\epsilon}_{m\cdot}(\mathbf{cl}((\mathbf{d}_1,\dots,\mathbf{d}_m)^\top)),$$

where $V(x) = Ax + b$ for all $x \in \mathbb{R}^n$ and $I \in \gamma(\overline{\mathbb{R}}^m)$, we add (according to Lemma 1 and the definition of $F^{V,I,\epsilon}$) the following linear constraints

$$\begin{aligned}
\mathbf{d}_i &\leq \mathbf{d}'_i + \epsilon_i . T_{i\cdot}(A(\mathbf{x}_{i,1},\dots,\mathbf{x}_{i,n})^\top + b) \wedge \alpha_{k\cdot}(I) && \forall i \in \{1,\dots,m\}\\
T_{i\cdot}(\mathbf{x}_{i,1},\dots,\mathbf{x}_{i,n}) &\geq \mathbf{d}'_i && \forall i \in \{1,\dots,m\}\\
T_{j\cdot}(\mathbf{x}_{i,1},\dots,\mathbf{x}_{i,n}) &\leq \mathbf{d}'_j && \forall i,j \in \{1,\dots,m\}\\
\mathbf{d}'_i &\leq T_{i\cdot}(\mathbf{x}'_{i,1},\dots,\mathbf{x}'_{i,m})^\top && \forall i \in \{1,\dots,m\}\\
T_{j\cdot}(\mathbf{x}'_{i,1},\dots,\mathbf{x}'_{i,n}) &\leq \mathbf{d}_j && \forall i,j \in \{1,\dots,m\}
\end{aligned}$$

where $\mathbf{d}'_i$, $\mathbf{x}_{i,j}$, $\mathbf{x}'_{i,j}$ are fresh variables for all $i,j \in \{1,\dots,m\}$.

For every inequality constraint $\mathbf{d} \leq [\![\Xi]\!]^\sharp_{k\cdot}(\mathbf{d}_1,\dots,\mathbf{d}_m)$, where $\Xi = \{(x,y)^\top \in \mathbb{R}^{2n} \mid Ax \leq b\}$ with $A \in \mathbb{R}^{l\times 2n}$ and $b \in \mathbb{R}^l$, we add (according to Equation (2)) the following linear constraints:

$$\begin{aligned}
\mathbf{d} &\leq T_{k\cdot}(\mathbf{y}_1,\dots,\mathbf{y}_n)^\top & T_{i\cdot}(\mathbf{x}_1,\dots,\mathbf{x}_n)^\top &\leq \mathbf{d}_i \quad \forall i \in \{1,\dots,m\}\\
& & A_{i\cdot}(\mathbf{x}_1,\dots,\mathbf{x}_n,\mathbf{y}_1,\dots,\mathbf{y}_n)^\top &\leq b_i \quad \forall i \in \{1,\dots,l\}
\end{aligned}$$

Here, $\mathbf{x}_1,\dots,\mathbf{x}_n$ and $\mathbf{y}_1,\dots,\mathbf{y}_n$ are fresh variables. Finally, we get

$$\rho^\sigma(\mathbf{z}) = \sup\{\rho(\mathbf{z}) \mid \rho : \mathbf{X} \to \mathbb{R} \text{ and } \rho(\mathbf{x}) \leq [\![e]\!]\rho \text{ for all constraints } \mathbf{x} \leq e \text{ of } \sigma'\}$$

for all $\mathbf{z} \in \mathbf{X}$. Hence, for all $\mathbf{z} \in \mathbf{X}$, $\rho^\sigma(\mathbf{z})$ can be computed by solving a linear programming problem that can be constructed in polynomial time:

Lemma 8. *For every max-strategy σ of $\mathcal{C}(\Psi)$, the variable assignment ρ^σ defined by Equation (9) can be computed in polynomial time through linear programming. Whenever the max-strategy algorithm must compute $\mu_{\geq\rho}[\![\sigma]\!]$, we have $\mu_{\geq\rho}[\![\sigma]\!] = \rho^\sigma$.* □

Example 7. We consider the max-strategy σ_2 from Example 6, i.e., we aim at computing $\mu_{\geq\rho_1}[\![\sigma_2]\!]$ which equals ρ^{σ_2} as defined in Equation (9). Therefore, for all $i \in \{1,\dots,m\}$, we aim at solving the following optimization problem:

$$\begin{aligned} \sup\ \mathbf{d}_{1,i} \qquad & \mathbf{d}_{1,1} \leq F_{1.}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1},\mathbf{d}_{1,2},\mathbf{d}_{1,3})^\top)) \\ & \mathbf{d}_{1,2} \leq F_{2.}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}(\mathbf{cl}((\mathbf{d}_{1,1},\mathbf{d}_{1,2},\mathbf{d}_{1,3})^\top)) \qquad \mathbf{d}_{1,3} \leq 0 \end{aligned}$$

We are now going to simplify the constraints that define the feasible space. By unfolding the definition of $\mathbf{cl}$ and the definition of $F_{i.}^{\mathbf{D}(1),\mathbf{I}(1),\epsilon(1)}$ for all $i \in \{1,2\}$, we obtain:

$$\begin{aligned} &\mathbf{d}_{1,1} \leq \mathbf{d}'_{1,1} + 5 + \sup\{-x_1 \mid x_1 = \mathbf{d}'_{1,1}, x_2 \leq \mathbf{d}'_{1,2}, x_2 - x_1 \leq \mathbf{d}'_{1,3}\} \\ &\mathbf{d}_{1,1} \leq 2.5 \\ &\mathbf{d}_{1,2} \leq \mathbf{d}'_{1,2} + 1 + \sup\{0 \mid x_1 \leq \mathbf{d}'_{1,1}, x_2 = \mathbf{d}'_{1,2}, x_2 - x_1 \leq \mathbf{d}'_{1,3}\} \\ &\mathbf{d}_{1,3} \leq 0 \\ &\mathbf{d}'_{1,1} \leq \sup\{x_1 \mid x_1 \leq \mathbf{d}_{1,1}, x_2 \leq \mathbf{d}_{1,2}, x_2 - x_1 \leq \mathbf{d}'_{1,3}\} \\ &\mathbf{d}'_{1,2} \leq \sup\{x_2 \mid x_1 \leq \mathbf{d}_{1,1}, x_2 \leq \mathbf{d}_{1,2}, x_2 - x_1 \leq \mathbf{d}'_{1,3}\} \\ &\mathbf{d}'_{1,3} \leq \sup\{x_2 - x_1 \mid x_1 \leq \mathbf{d}_{1,1}, x_2 \leq \mathbf{d}_{1,2}, x_2 - x_1 \leq \mathbf{d}'_{1,3}\} \end{aligned}$$

After eliminating the supremums in the right-hand sides, we get:

$$\begin{array}{llllll} \mathbf{d}_{1,1} \leq \mathbf{d}'_{1,1} + 5 - \mathbf{x}_{1,1,1} & \mathbf{x}_{1,1,1} = \mathbf{d}'_{1,1} & \mathbf{x}_{1,1,2} \leq \mathbf{d}'_{1,2} & \mathbf{x}_{1,1,2} - \mathbf{x}_{1,1,1} \leq \mathbf{d}'_{1,3} \\ \mathbf{d}_{1,1} \leq 2.5 & & & \\ \mathbf{d}_{1,2} \leq \mathbf{d}'_{1,2} + 1 & \mathbf{x}_{1,2,1} \leq \mathbf{d}'_{1,1} & \mathbf{x}_{1,2,2} = \mathbf{d}'_{1,2} & \mathbf{x}_{1,2,2} - \mathbf{x}_{1,2,1} \leq \mathbf{d}'_{1,3} \\ \mathbf{d}_{1,3} \leq 0 & & & \\ \mathbf{d}'_{1,1} \leq \mathbf{x}'_{1,1,1} & \mathbf{x}'_{1,1,1} \leq \mathbf{d}_{1,1} & \mathbf{x}'_{1,1,2} \leq \mathbf{d}_{1,2} & \mathbf{x}'_{1,1,2} - \mathbf{x}'_{1,1,1} \leq \mathbf{d}_{1,3} \\ \mathbf{d}'_{1,2} \leq \mathbf{x}'_{1,2,2} & \mathbf{x}'_{1,2,1} \leq \mathbf{d}_{1,1} & \mathbf{x}'_{1,2,2} \leq \mathbf{d}_{1,2} & \mathbf{x}'_{1,2,2} - \mathbf{x}'_{1,2,1} \leq \mathbf{d}_{1,3} \\ \mathbf{d}'_{1,3} \leq \mathbf{x}'_{1,3,2} - \mathbf{x}'_{1,3,1} & \mathbf{x}'_{1,3,1} \leq \mathbf{d}_{1,1} & \mathbf{x}'_{1,3,2} \leq \mathbf{d}_{1,2} & \mathbf{x}'_{1,3,2} - \mathbf{x}'_{1,3,1} \leq \mathbf{d}_{1,3} \end{array}$$

When we solve the corresponding linear programming problems that aim at maximizing $\mathbf{d}_{1,1}$, $\mathbf{d}_{1,2}$, and $\mathbf{d}_{1,3}$, respectively, we get $\mathbf{d}_{1,1} = 2.5$, $\mathbf{d}_{1,2} = 3.5$, and $\mathbf{d}_{1,3} = 0$. Observe that, instead of solving three linear programming problems, we could solve just one, where we aim at maximizing the sum $\mathbf{d}_{1,1} + \mathbf{d}_{1,2} + \mathbf{d}_{1,3}$. Any optimal solution then gives us the values for $\mathbf{d}_{1,1}$, $\mathbf{d}_{1,2}$, and $\mathbf{d}_{1,3}$. □

The Final Results. Putting everything together, we finally get:

Theorem 3. *The abstract semantics of an affine hybrid automaton $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$ can be computed through the presented max-strategy improvement algorithm. This algorithm performs at most exponentially many strategy improvement steps, each*

of which can be performed in polynomial time through linear programming. The number of max-strategy improvement steps is bounded by

$$m \cdot |\mathbf{L}| \cdot \prod_{l \in \mathbf{L}} \max\{1, |\{(l_1, \rho, l_2) \in \mathcal{T} \mid l_2 = l\}|^m\},$$

where m denotes the number of linear templates. The time elapse operation for affine hybrid automata w.r.t. template polyhedra can be performed in polynomial time. □

Theorem 4. *Abstract reachability for affine hybrid systems w.r.t. template polyhedra is in* co−NP.

Proof. Let $\Psi = (n, \mathbf{L}, \mathcal{T}, \Theta, \mathbf{D}, \mathbf{I}, l_0)$ be an affine hybrid system, $l \in \mathbf{L}$, and $T \in \mathbb{R}^{m \times n}$ a template constraint matrix. We have to provide an non-deterministic algorithm that has an accepting run iff $\mu[\![\mathcal{C}(\Psi)]\!](\mathbf{d}_{l,i}) = -\infty$ for all $i \in \{1, \ldots, m\}$, i.e., l is unreachable.

The algorithm firstly chooses a max-strategy σ for $\mathcal{C}(\Psi)$ non-deterministically. Note that there exists a max-strategy σ' for $\mathcal{C}(\Psi)$ such that $\rho^{\sigma'} = \mu[\![\mathcal{C}(\Psi)]\!]$. According to Lemma 8, we then compute ρ^σ as defined by Equation (9) in polynomial time. If ρ^σ solves $\mathcal{C}(\Psi)$ (this can be checked in polynomial time), then we know that $\rho^\sigma \geq \mu[\![\mathcal{C}(\Psi)]\!]$. Hence, the algorithm accepts iff $\rho^\sigma(\mathbf{d}_{l,i}) = -\infty$ for all $i \in \{1, \ldots, m\}$. □

5 Conclusion

In this paper we showed how the max-strategy improvement algorithm of Gawlitza and Seidl [10, 11, 12, 14, 15] can be utilized to compute abstract semantics of affine hybrid automata w.r.t. template polyhedra — an problem that can be used for *unbounded time verification*. This gives us a polynomial-time algorithm for the time elapse operation over template polyhedra. Moreover, we showed that the corresponding abstract reachability problem is in co−NP. For future work, it would be interesting to see in how far this approach can be generalized to non-linear templates and non-linear dynamics (cf. Gawlitza and Seidl [14]). It also remains to evaluate the proposed approach in practice. We report on our proof-of-concept implementation in the corresponding technical report Dang and Gawlitza [8].

References

[1] Adjé, A., Gaubert, S., Goubault, E.: Computing the smallest fixed point of nonexpansive mappings arising in game theory and static analysis of programs. In: MTNS (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)
[3] Alur, R., Dang, T., Ivancic, F.: Counter-example guided predicate abstraction of hybrid systems. Theoretical Computer Science (TCS) 354(2), 250–271 (2006)
[4] Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Inf. 43(7), 451–476 (2007)
[5] Clarke, E.M., Fehnker, A., Han, Z., Krogh, B.H., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. 14(4), 583–604 (2003)

[6] Costan, A., Gaubert, S., Goubault, É., 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] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
[8] Dang, T., Gawlitza, T.M.: Template-based unbounded time verification of affine hybrid automata. Technical report, VERIMAG, Grenoble, France (2011)
[9] Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static analysis by policy iteration on relational domains. In: Nicola [17]
[10] 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)
[11] Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: Nicola [17]
[12] Gawlitza, T., Seidl, H.: Precise Interval Analysis vs. Parity Games. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 342–357. Springer, Heidelberg (2008)
[13] Gawlitza, T.M., Monniaux, D.: Improving Strategies via SMT Solving. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 236–255. Springer, Heidelberg (2011)
[14] Gawlitza, T.M., Seidl, H.: Computing Relaxed Abstract Semantics w.r.t. Quadratic Zones Precisely. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 271–286. Springer, Heidelberg (2010)
[15] Gawlitza, T.M., Seidl, H.: Solving systems of rational equations through strategy iteration. TOPLAS (accepted, to appear)
[16] Jurdzinski, M.: Deciding the winner in parity games is in up ∩ co-up. Inf. Process. Lett. 68(3), 119–124 (1998)
[17] De Nicola, R. (ed.): ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
[18] Prajna, S., Jadbabaie, A.: Safety Verification of Hybrid Systems using Barrier Certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
[19] 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)
[20] Sankaranarayanan, S., Dang, T., Ivančić, F.: A Policy Iteration Technique for Time Elapse over Template Polyhedra. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 654–657. Springer, Heidelberg (2008)
[21] Tarski, A.: A lattice-theoretical fixpoint theorem and its appications. Pac. J. Math. 5, 285–309 (1955)
[22] Tiwari, A., Khanna, G.: Series of Abstractions for Hybrid Automata. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 465–478. Springer, Heidelberg (2002)
[23] 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)

Access-Based Localization with Bypassing

Hakjoo Oh and Kwangkeun Yi

Seoul National University

Abstract. We present an extension of access-based localization technique to mitigate a substantial inefficiency in handling procedure calls. Recently, access-based localization was proposed as an effective way of tightly localizing abstract memories. However, it has a limitation in handling procedure calls: the localized input memory for a procedure contains not only memory locations accessed by the procedure but also those accessed by transitively called procedures. The weakness is especially exacerbated in the presence of recursive call cycles, which is common in analysis of realistic programs. In this paper, we present a technique, called bypassing, that mitigates the problem. Our technique localizes input memory states only with memory locations that the procedure directly accesses. Those parts not involved in analysis of the procedure are bypassed to transitively called procedures. In experiments with an industrial-strength global C static analyzer, the technique reduces the average analysis time by 42%. In particular, the technique is especially effective for programs that extensively use recursion: it saves analysis time by 77% on average.

1 Introduction

Memory localization is vital for reducing global analysis cost [12,14,3,23,22]. The performance problem of flow-sensitive global analysis is that code blocks such as procedure bodies are repeatedly analyzed (often needlessly) with different input memory states. Localization of input abstract memories, which removes the irrelevant memory entries that will not be used inside called procedure bodies, alleviates the problem by increasing the chance of reusing previously computed analysis results. For example, consider a code `x=0;f();x=1;f();` and assume that `x` is not used inside `f`. Without localization, `f` is analyzed twice because the input state to `f` is changed at the second call. If `x` is removed from input states (localization), the analysis result for the first call can be reused at the second call without re-analyzing the procedure body.

Access-based technique provides an effective way of realizing memory localization [14]. Because localization must be done before analyzing a procedure, it is impossible to exactly compute to-be-used parts of input memory. Thus, some approximation must be involved, so that the localized state can contain some spurious entries that will not be actually used by the procedure. The conventional approximation methods are reachability-based techniques: from input memory, only the abstract locations reachable from actual parameters and global locations are collected. However, the technique is too conservative in practice because

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 50–65, 2011.

only few reachable locations are actually accessed [14]. Access-based technique, on the other hand, trims input memory states more aggressively: locations that are reachable but may not be accessed are additionally removed. The access information is computed by a conservative pre-analysis. Thus, access-based localization more effectively reduces global analysis cost than the reachability-based technique does [14].

However, the localization has a source of inefficiency in handling procedure calls. In access-based localization[1], the localized input state for a procedure involves not only the abstract locations that are accessed by the called procedure but also those locations that are accessed by transitively called procedures. For instance, when procedure f calls g, the localized state for f contains abstract locations that are accessed by g as well as abstract locations accessed by f. Those locations that are exclusively accessed by g are, however, irrelevant to the analysis of f because they are not used in analyzing f. Even so, those locations are involved in the localized state (for f), which sometimes leads to unnecessary computational cost (due to re-analyses of procedure body).

Such inefficiency is especially exacerbated with recursive call cycles. Consider a recursive call cycle $f \to g \to h \to f \to \cdots$. Because of the cyclic dependence among procedures, every procedure receives input memories that contain all abstract locations accessed by the whole cycle. That is, access-based localization does not help any more inside call cycles. Moreover, recursive cycles (even large ones) are common in real C programs. For example, in GNU open source code, we noticed that a number of programs have large recursive cycles and a single cycle sometimes contains more than 40 procedures. This is the main performance bottleneck of access-based localization in practice (Section 4.2).

In this paper, we extend access-based localization technique so that the aforementioned inefficiency can be relieved. With our technique, localized states for a procedure contains only the abstract locations that are accessed by the procedure and does not contain other locations that are exclusively accessed by transitively called procedures. Those excluded abstract locations are "bypassed" to the transitively called procedures, instead of passing through the called procedure. In this way, analysis of a procedure involves only the memory parts that the procedure directly accesses (even inside recursive cycles), which results in more tight localization and hence reduces analysis cost more than access-based localization does. The following example illustrates how our technique saves cost.

Example 1. Consider the following code.

```
1: int a=0, b=0;
2: void g() { b++; }
3: void f() { a++; g(); }
4: int main () {
5:     b=1; f();       // first call to f
6:     b=2; f(); }     // second call to f
```

[1] In fact, any localization techniques suffers from similar problems. In this paper, we discuss the problem in the context of access-based localization.

Procedure `main` calls `f`, and `f` calls `g`. Procedures `f` and `g` update the value of `a` and `b`, respectively. Procedure `main` calls `f` two times with the value of `b` changed.

- With access-based localization: Both `f` and `g` are analyzed two times. The localized input memory for `f` at the first call (line 5) contains locations `a` and `b` because both are (directly/indirectly) accessed while analyzing `f`. The localized state at the second call (line 6) contains the same locations. Because the value of `b` is changed, `f` (as well as `g`) is re-analyzed at the second call.
- With our technique: `f` is analyzed only once (though `g` is analyzed twice). Localized memories for procedure `f` contain only the location that `f` directly accesses, i.e., `a`. The value of `a` is not changed and the body of `f` is not re-analyzed at the second call. However, procedure `g` is re-analyzed because we propagate the changed value of `b` to the entry of `g`.

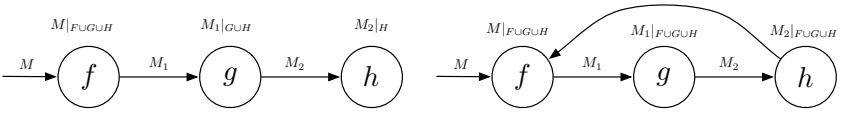
In experiments with an industrialized abstract interpretation-based static analyzer, our technique saved 9–79%, on average 42%, in analysis time in comparison with the access-based localization technique for a variety of open-source C benchmarks (2K–100K). In particular, for those benchmark programs that extensively use recursion and have large recursive call cycles, our technique is more effective: it reduces the analysis time for those programs by 77% on average. The technique does not compromise the analysis precision.

Contributions. This paper makes the following contributions.

- We report on a substantial performance degradation of localization and present a technique to mitigate the problem. Our technique is meaningful because real C programs often have complex procedural relationships such as large recursive cycles that significantly exacerbate the problem. Though we focus on access-based localization, any localization schemes (including reachability-based ones) suffer from similar (basically the same) problems. To the best of our knowledge, these aspects of localization techniques have not been adequately addressed in the literature.
- We prove the effectiveness of our technique by experiments with an industrial-strength C static analyzer [8,9,10,13,14,15].

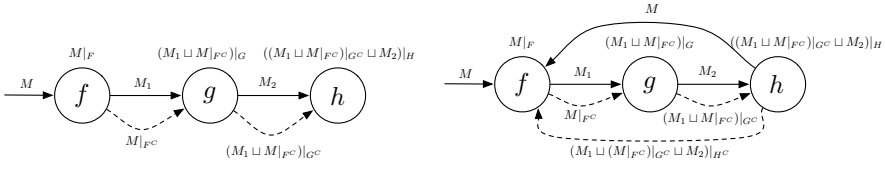
Overview. We illustrate how our technique works with examples. Fig. 1 shows example call graphs. There are three procedures: f, g and h. Suppose F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. We describe how the problem occurs and then how to overcome the problem.

Access-based localization has inefficient aspects in analyzing procedure calls. We first consider the case for non-recursive call chains (Fig. 1(a)). With the localization, the input memory M to f is localized so that the procedure f is analyzed only with a subpart $M|_{F \cup G \cup H}$ (M with projected on locations set $F \cup G \cup H$) rather than the entire input memory. Similarly, the input memory M_1 to g is localized to $M|_{G \cup H}$, and h's input memory M_2 is localized to $M_2|_H$. The inefficiency comes from the fact that not the entire localized memory is accessed



(a) Non-recursive call chain (b) Recursive call cycle

Fig. 1. Problem of localization. F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. $M|_F$ denotes the memory state M with projected on abstract locations F.



(a) Non-recursive call chain (b) Recursive call cycle

Fig. 2. Illustration of our technique. With our technique, each procedure is analyzed with its respective directly accessed locations, and others are bypassed (dashed line) to the subsequent procedure.

by each procedure. For example, abstract locations $G \cup H$ are not necessary in analyzing the body of f.

The problem becomes severe when analyzing recursive call cycles. Consider Fig. 1(b). As in the previous case, the input memory M to f is localized to $M|_{F\cup G\cup H}$. However, in this case, the input memory M_1 to g is also projected on $F \cup G \cup H$, not on $G \cup H$, because f can be called from g through the recursive cycle. Similarly, input memory M_2 to h is localized to $M|_{F\cup G\cup H}$. In summary, localization does not work any more inside the cycle.

Fig. 2 illustrates how our technique works. We first consider non-recursive call case (Fig. 2(a)). Instead of restricting f's input memory to $F \cup G \cup H$, we localize it with respect to only the directly accessed locations, i.e., F. Thus, f is analyzed with $M|_F$. The non-localized memory part ($M|_{F^C}$) is directly bypassed (dashed line) to g. Then, the output memory M_1 from f and the bypassed memory $M|_{F^C}$ are joined to prepare input memory $M_1 \sqcup M|_{F^C}$ for procedure g. The input memory is localized to $(M_1 \sqcup M|_{F^C})|_G$ and g is analyzed with the localized memory. Again, the non-localized parts $(M_{F^C} \sqcup M_1)|_{G^C}$ are bypassed to the subsequent procedure h. In this way, each procedure is analyzed only with abstract locations that the procedure directly accesses.

The technique is naturally applicable to recursive cycles (Fig. 2(b)). With our technique, even procedures inside recursive call cycles are analyzed with memory parts that are directly accessed by each procedure. Hence, in Fig. 2(b), the localized memory for f (resp., g and h) only contains locations F (resp., G and H).

2 Setting: Analysis Framework

We describe our analysis framework. The analysis basically performs flow-sensitive and context-insensitive global analysis, and an abstract memory state is represented by a map from abstract locations to abstract values. In Section 3, we present our technique on top of this framework. Section 2.1 shows the intermediate representation of programs, and Section 2.2 defines the analysis in terms of abstract domain and semantics.

2.1 Graph Representation of Programs

We assume that a program is represented by a supergraph [17]. A supergraph consists of control flow graphs of procedures with interprocedural edges connecting each call-site to its callee and callees to the corresponding return-sites. Each node $n \in Node$ in the control flow graph has one of the four types :

$$entry \mid exit \mid call(f_x, e) \mid return \mid set(lv, e)$$

Each control flow graph has *entry* and *exit* nodes. A call-site in a program is represented by a call node and its corresponding return node. A call node $call(f_x, e)$ indicates that it invokes a procedure f, its formal parameter is x, and the actual parameter is e. For simplicity, we assume that there are no function pointers in the program and consider only one parameter. Node type *return* indicates a return node of a call node. $set(lv, e)$ represents an assignment statement that assigns the value of e into the location that l-value expression lv denotes. In this paper, we do not restrict expression (e) and l-value expression (lv) to specific ones. We assume that edges in flow graphs are assembled by function $\mathsf{succof} \in Node \to 2^{Node}$, which maps each node to its successors.

2.2 Static Analysis

We consider static analyses, in which the set of (possibly infinite) concrete memory states are represented by an abstract memory state:

$$\hat{Mem} = \hat{Addr} \to \hat{Val}$$

That is, $\hat{Mem}$ is a map from abstract locations ($\hat{Addr}$) to the abstract values ($\hat{Val}$). We assume that $\hat{Addr}$ is a finite set and $\hat{Val}$ is an arbitrary cpo (complete partial order). We assume further that abstract values and locations are computed by two functions $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$, respectively. Given an expression e and an abstract memory state $\hat{m}$, $\hat{\mathcal{V}}(\in e \to \hat{Mem} \to \hat{Val})$ evaluates the abstract value that e denotes under $\hat{m}$. Similarly, $\hat{\mathcal{L}}(\in lv \to \hat{Mem} \to 2^{\hat{Addr}})$ evaluates the set of abstract locations of lv under $\hat{m}$.

With $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$, we define semantic function $\hat{f} : Node \to \hat{Mem} \to \hat{Mem}$. Given a node n and an input memory state m, $\hat{f}(n)(m)$ computes the effect of the command in node n on the input state :

$$\hat{f}(n)(\hat{m}) = \begin{cases} \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \hat{\mathcal{L}}(x)(\hat{m})\} & \text{if } n = call(f_x, e) \\ \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \hat{\mathcal{L}}(lv)(\hat{m})\} & \text{if } n = set(lv, e) \\ \hat{m} & \text{otherwise} \end{cases}$$

where, $\hat{m}\{v /\!\!/ \{l_1, \ldots, l_k\}\}$ means $\hat{m}\{l_1 \mapsto (\hat{m}(l_1) \sqcup v)\} \cdots \{l_k \mapsto (\hat{m}(l_k) \sqcup v)\}$. The effect of node $set(lv, e)$ is just to (weakly) assign the abstract value of e into the locations in $\hat{\mathcal{L}}(lv)(\hat{m})$.[2] The call node command $call(f_x, e)$ binds the formal parameter x to the value of actual parameter e. Please note that the output of the call node is the memory state that flows into the body of the called procedure, not the memory state returned from the call.

Then, the analysis is to compute a fixpoint table $\mathcal{T} \in Node \to \hat{Mem}$ that maps each node in the program to its (input) abstract memory state. The map is defined by the least fixpoint of the following function $\hat{F}$:

$$\hat{F} : (Node \to \hat{Mem}) \to (Node \to \hat{Mem})$$
$$\hat{F}(T) = \lambda n. \bigsqcup_{p \in \mathsf{predof}(n)} \hat{f}(n)(\mathcal{T}(p))$$

The fixpoint is computed by a worklist algorithm. The worklist consists of nodes of the control flow graph of the program whose abstract state has to be recomputed. When a computed memory state for n is changed, we add successors of n into the worklist. The algorithm uses widening operation [2] to guarantee termination. Fig. 4(a) shows the algorithm.

3 Access-Based Localization with Bypassing

In this section, we describe our technique on top of the analysis framework (Section 2). Our technique is an extension of the access-based localization. In Section 3.1, we describe the access-based localization. Then, we extends the localization technique to derive our bypassing technique.

3.1 Access-Based Localization: Previous Approach

In access-based localization [14], the entire analysis is staged into two phases: (1) a pre-analysis conservatively estimates the set of abstract locations that will be accessed during actual analysis for each procedure; (2) then, the actual analysis uses the access-set results and, right before analyzing each procedure, filters out memory entries that will not be accessed inside the procedure's body.

The pre-analysis is a further abstraction of the original analysis. The pre-analysis must be safe in that the estimated access information should be conservative with respect to the actual access set that would be used during the actual analysis. Moreover, to be useful, the estimation should be efficient enough to compensate for the extra burden of running pre-analysis once more. In [14],

[2] For brevity, we consider only weak updates. Applying strong update is orthogonal to our technique we present in this paper.

such a pre-analysis is obtained by applying conservative abstractions (such as ignoring statement orders, i.e., flow-insensitivity) to the abstract semantics of original analysis. During the pre-analysis, the access-sets for each program point are collected. Let $\mathcal{A} \in \mathit{Node} \to 2^{A\hat{d}dr}$ be the access information. That is, all the abstract locations that are accessed during the analysis at node n are represented by $\mathcal{A}(n)$.

The actual analysis is the same as the original analysis except for localizing operation. Because actual analysis additionally performs localizations using the access-set information, the abstract semantics for call node is changed. Thus, in actual analysis, non-accessed memory locations are excluded from the input memories of procedures: given an input memory state $\hat{m}$ to a call node $\mathit{call}(f_x, e)$, the semantic function $\hat{f}$ for the call statement $\mathit{call}(f_x, e)$ is changed as follows:

$$\hat{f}\ \mathit{call}(f_x, e)\ \hat{m} \quad = \quad \hat{m}'|_{\mathsf{access}(f)} \text{ where } \hat{m}' = \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \{x\}\}$$

That is, after parameter binding ($\hat{m}'$) the memory is restricted on $\mathsf{access}(f)$, where $\mathsf{access}(f)$ is defined as follows: ($\mathsf{callees}(f)$ denotes the set of procedures, including f, that are reachable from f via the call-graph and $\mathsf{nodesof}(f)$ the set of nodes in procedure f.)

$$\mathsf{access}(f) = \bigcup_{g \in \mathsf{callees}(f)} (\bigcup_{n \in \mathsf{nodesof}(g)} \mathcal{A}(n))$$

$\mathsf{access} \in \mathit{ProcId} \to 2^{A\hat{d}dr}$ maps each procedure to a set of abstract locations that are possibly accessed during the analysis of callee procedures. Note that we consider all the transitively called procedures as well, instead of just considering the directly called procedure.

3.2 Access-Based Localization with Bypassing: Our Approach

Definition 1 (directly/indirectly(transitively) called procedure). *When a procedure f is called from a call-site, we say that f is a directly called procedure from the call-site, and procedures that are reachable from f via the call-graph are indirectly (or transitively) called procedures.*

Example 2. Consider a call chain $f \to g \to h$. When f is called from a call-site, f is the directly called procedure, and g and h are indirectly called procedures.

Definition 2 (directly/indirectly accessed locations). *When a procedure f is called from a call-site, we say that a location is directly accessed by procedure f if the location is accessed inside the body of f. We say that the location is indirectly accessed by f if the location is not accessed inside f's body but accessed by indirectly called procedures.*

Example 3. Consider a call chain $f \to g$, and assume that locations l_1 is accessed inside the body of f and l_2 is accessed inside the body of g. We say l_1 is directly accessed by f and l_2 is indirectly accessed by f (l_2 is directly accessed by g).

Our technique is an extension of access-based localization. Thus, we also separate the entire analysis into two phases: pre-analysis, and actual analysis.

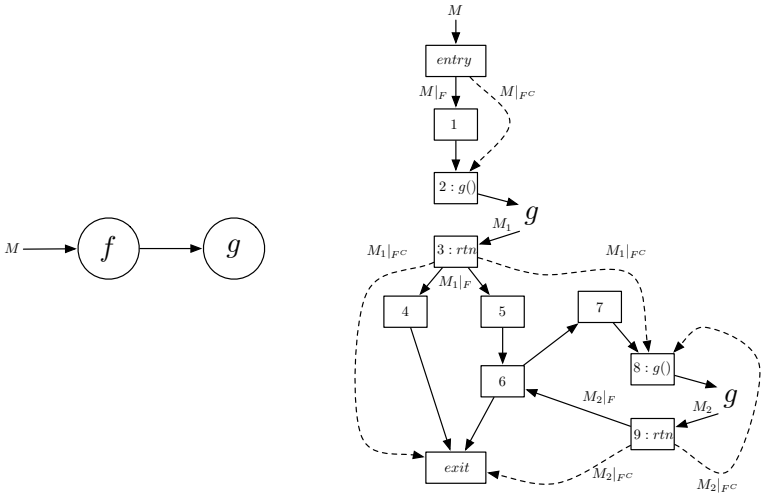
Pre-analysis is slightly changed. Pre-analysis is exactly the same with the one that would be used in access-based localization, except that we use its result in a different way. In access-based localization, we compute $\mathsf{access}(f)$, which includes abstract locations directly accessed by f as well as locations indirectly accessed by f. Instead, our technique computes $\mathsf{direct} \in \mathit{ProcId} \to 2^{\hat{Addr}}$ that maps each procedure to a set of abstract locations that are directly accessed by the procedure, excluding indirectly accessed locations. Given $\mathcal{A} : \mathit{Node} \to 2^{\hat{Addr}}$ from pre-analysis, the set $\mathsf{direct}(f)$ is defined as follows:

$$\mathsf{direct}(f) = \bigcup_{n \in \mathsf{nodesof}(g)} \mathcal{A}(n)$$

Major changes are in actual analysis. With access-based localization, actual analysis performs localization using the access information from pre-analysis. Now, the actual analysis is changed in two ways: the analysis performs the localization in a different way, and it additionally performs another technique, called bypassing. When analyzing a procedure, we localize the input memory state so that only the abstract locations directly accessed by the procedure are passed to the current procedure. The non-localized parts, which contains indirectly accessed locations, are not passed to the directly called procedure but bypassed to indirectly called procedures. In this way, every procedure is analyzed with input memory state that is more tightly localized than access-based localization. In terms of analysis on control flow graphs, these operations work as follows:

– **Localization:** Localization is performed at nodes where memory states flow into the nodes from other procedures. These nodes include entry and return nodes: when a procedure is called from a call-site, the input memory from the call-site flows into entry of the called procedure, and when a procedure returns, the memory state returned from the procedure flows into its caller via a return node. Hence, the memory states at entry and return nodes of a procedure are localized so that the procedure is analyzed with the directly accessed locations. We call such nodes, where localization occurs, bypassing sources.

  Example 4. Consider the Fig. 3. Fig. 3(a) shows a call-graph, where procedure f calls g, and Fig. 3(b) shows the control flow graph for f. Let F and G be the set of abstract locations that are directly accessed by procedure f and g, respectively. There are three bypassing sources: *entry*, node 3, and node 9. Nodes 3 and 9 are return nodes. At *entry*, the input memory M is restricted on F. Hence, node 1 is analyzed with the localized memory $M|_F$. At node 3 and 9, the memory returned from procedure g, M_1 and M_2 are restricted on the location set F, and hence, the body of procedure f is always analyzed with the local memory $M|_F$. By contrast, with access-based localization, f is analyzed with the localized memory $M|_{F \cup G}$, which is strictly bigger than $M|_F$.



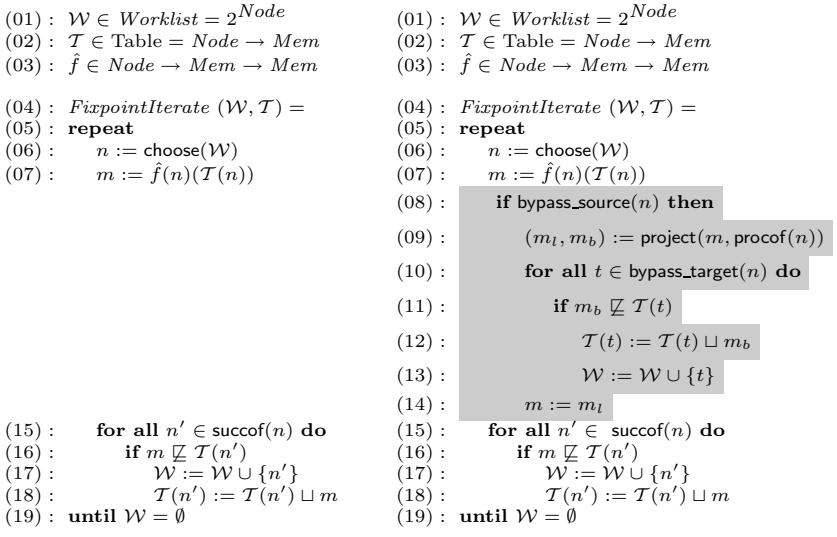
(a) call-graph (b) control flow graph of f

Fig. 3. Example: (a) a call-graph, where f is called with input memory state M and g is called from f (b) inside view (control flow graph) of f, where solid lines represent control flow edges and dashed lines represent bypassing edges

– **Bypassing:** Bypassing happens between bypassing sources and targets. The non-localized parts at bypassing sources (entry or return nodes) should be delivered to nodes where memory states flow into other procedures. These nodes include procedure exit and call nodes: at procedure exit, the output memory state of the procedure is propagated to the caller, and at call nodes, memory states flow into called procedures. Thus, after performing localization at a bypassing source, the non-localized parts are bypassed to "immediate" call or exit nodes that are reachable without passing through other call nodes. We call such call and exit nodes as bypassing targets.

  Example 5. Consider the Fig. 3(b) again. The solid lines represent control flow graphs of procedure f and dashed lines shows how bypassing happens. There are three bypassing sources: *entry*, 3, and 9. The bypassing target for *entry* is the call node 2. Another call node 8 or exit node are not bypassing target for *entry* because they are not reachable from *entry* without passing through the call node 2. And, bypassing targets for node 3 are 8 and *exit*. Similarly, bypassing targets for node 9 are 8 and *exit*. At entry node, the non-localized memory parts ($M|_{F^C}$) are bypassed to entry's bypassing target, node 2. Similarly, at nodes 3 and 9, the non-localized memory $M_1|_{F^C}$ and $M_2|_{F^C}$ are bypassed to their bypassing targets, node 8 and *exit*.

Fig. 4(b) shows our technique integrated in the worklist-based analysis algorithm. In order to transform access-based localization into our technique, only shaded lines are inserted; other parts remain the same. Predicate bypass_source $\in$



(a) The worklist-based algorithm (b) The algorithm with bypassing

Fig. 4. Comparison of the normal analysis algorithm and our bypassing algorithm: our technique is a simple addition of the traditional algorithm

$Node \to bool$ checks whether a node is a bypass source or not. Function $\mathsf{procof} \in Node \to ProcId$ gives name of the procedure that encloses the given node. Function project takes a memory state and a procedure and partitions the input memory into directly accessed and indirectly accessed parts:

$$\mathsf{project}(m, f) = (m|_{\mathsf{direct}(f)}, m|_{\mathsf{access}(f) \setminus \mathsf{direct}(f)})$$

Function $\mathsf{bypass\_target} \in Node \to 2^{Node}$ maps each bypass source to its bypass targets. If the current node n is a bypass source (line 8), the memory state m is divided into a local memory m_l and the rest part m_b (line 9). The local memory m_l is propagated to the successors of n as in the case of the normal algorithm (line 14). The non-localized memory (m_b) is updated to the input memories of bypassing targets of n (line 10–13).

3.3 Delivery Points Optimization

Bypassing operation induces additional join operations, one of the most expensive operation in semantic-based static analyses [1,9]. At bypassing targets, the bypassed memory from the bypassing source should be joined with the memory propagated along usual control flows. For example, consider Fig. 3. At node 2, two input memories, one propagated from node 1 and another bypassed from *entry*, are joined. Similarly, at the other bypassing targets (node 8 and *exit*), additional join operations take place.

```
if (cond1) f1 ();
if (cond2) f2 ();
...
if (condk) fk ();
```

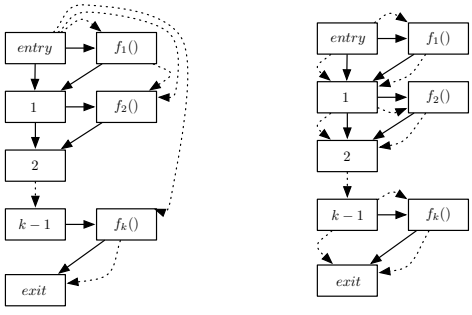


Fig. 5. Example of common code patterns that increases bypassing overhead

We noticed that the number of additional joins is sometimes unbearable. For example, Fig. 5 shows a common programming pattern: the left-hand shows the code pattern, and the middle shows its control flow graph with bypassing edges (dashed lines). Procedures `f1, f2,`$\cdots$`,fk` are sequentially called after respective condition checks (`cond1, cond2,`$\cdots$`, condk`). For this code, bypassing happens as follows (as dashed lines in Fig. 5 show):

- From *entry* to $f_1, f_2, f_3, \ldots, f_k, exit$
- From f_1 to $f_2, f_2, f_3, \ldots, f_k, exit$
- $\cdots$
- From f_k to *exit*

Thus, the total number of bypassing edges for this code fragment is $(k+1)(k+2)/2$ when k is the number of branches.

We mitigate the overhead by making bypassing pass through some particular nodes that reduces the total number of bypassing edges. These nodes, we call them "delivery points", include some join points and loop heads. For example, in Fig. 5, we use nodes $\{1, 2, \cdots, k-1\}$ as delivery points and let bypassing drop by those nodes. As a result, bypassing happens as shown in the rightmost graph in Fig. 5. Bypassing from *entry* to $call_1$ takes place as in before, but Instead of bypassing from *entry* to $\{f_2, \cdots, f_k, exit\}$, we pass through node $1, 2, \cdots, k-1$, which reduces the total number of bypassing edges from $(k+1)(k+2)/2$ to $3k$. In order to select such delivery points, we use a simple heuristic that uses join points or loop heads as delivery points when the selection actually reduces the total number of bypassing edges.

4 Experiments

We check the performance of our technique by experiments with Airac, a global abstract interpretation engine in an industrialized bug-finding analyzer Sparrow [8,9,10,13,14,15].

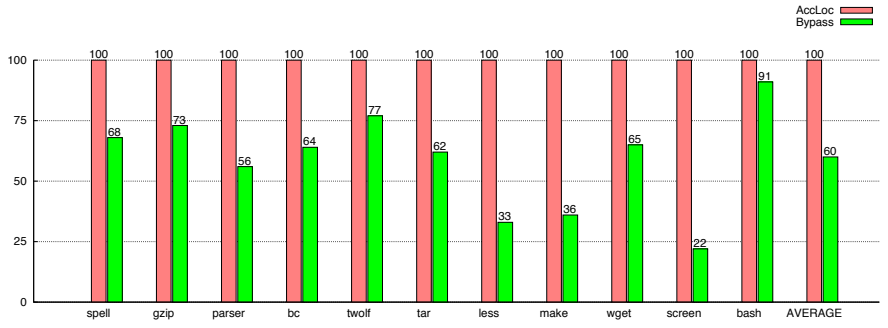


Fig. 6. Comparison of analysis time between access-based localization and bypassing

4.1 Setting Up

Airac is an interval-domain-based abstract interpreter. The analyzer performs flow-sensitive and context-insensitive global analysis: it computes a map $\mathcal{T} \in Node \to \hat{Mem}$ from program points ($Node$) to abstract memories ($\hat{Mem}$). An abstract memory is a map $\hat{Mem} = \hat{Addr} \to \hat{Val}$ where $\hat{Addr}$ denotes abstract locations that are either program variables or allocation sites, and $\hat{Val}$ denotes abstract values including interval values, addresses, array blocks, and structure blocks. The details of abstract domain and semantics are described in [14].

From our baseline analyzer Airac, we have made two analyzers: $\text{Airac}_{\text{AccLoc}}$ and $\text{Airac}_{\text{Bypass}}$ that respectively use the access-based localization and our technique. $\text{Airac}_{\text{Bypass}}$ is exactly the same as $\text{Airac}_{\text{AccLoc}}$ except that $\text{Airac}_{\text{Bypass}}$ additionally performs the bypassing operation. Hence, performance differences, if any, between them, are solely attributed to the bypassing technique. The analyzers are written in OCaml.

We have analyzed 10 software packages. Fig. 1 shows our benchmark programs. **LOC** indicates the number of lines of code before preprocessing. **Proc** indicates the number of procedures in each program. **LRC** represents the size of largest recursive call cycle contained in each program. For example, the program `screen` have 589 procedures and, among them, 77 procedures belong to a single recursive cycle. We analyzed each program globally: the entire program is analyzed starting from the entry of the main procedure. All experiments were done on a Linux 2.6 system running on a Pentium4 3.2 GHz box with 4 GB of main memory.

We use two performance measures: (1) *time* is the CPU time (in seconds) spent during the analysis; (2) *MB* is the peak memory consumption (in megabytes) during the analysis.

4.2 Results

Fig. 6 compares the *time* of $\text{Airac}_{\text{AccLoc}}$ and $\text{Airac}_{\text{Bypass}}$. Table 1 shows the raw analysis results. Overall, $\text{Airac}_{\text{Bypass}}$ saved 8.9%–78.5%, on average 42.1%, of the analysis time of $\text{Airac}_{\text{AccLoc}}$. There are some noteworthy points.

Table 1. Program properties and analysis results. Lines of code (**LOC**) are given before preprocessing. The number of procedures (**Proc**) is given after preprocessing. **LRC** represents the size of largest recursive call cycle contained in each program. *time* shows analysis time in seconds. *MB* shows peak memory consumption in megabytes. $\mathsf{Airac}_{\mathsf{AccLoc}}$ uses access-based localization for procedure calls and $\mathsf{Airac}_{\mathsf{Bypass}}$ uses our technique. *time* for $\mathsf{Airac}_{\mathsf{AccLoc}}$ and $\mathsf{Airac}_{\mathsf{Bypass}}$ are the total time that includes pre-analysis time. *Save* shows time savings in percentage of $\mathsf{Airac}_{\mathsf{Bypass}}$ against $\mathsf{Airac}_{\mathsf{AccLoc}}$.

Program	**LOC**	**Proc**	**LRC**	$\mathsf{Airac}_{\mathsf{AccLoc}}$		$\mathsf{Airac}_{\mathsf{Bypass}}$		*Save*
				time(sec)	*MB*	*time*(sec)	*MB*	(*time*)
spell-1.0	2,213	31	0	2.4	10	1.6	10	31.6%
gzip-1.2.4a	7,327	135	2	51.9	65	37.7	64	27.4%
parser	10,900	325	3	571.6	206	319.4	245	44.1%
bc-1.06	13,093	134	1	496.9	131	318.4	165	35.9%
twolf	19,700	192	1	509.5	212	389.9	212	23.5%
tar-1.13	20,258	222	13	2,407.9	294	1,503.2	338	37.6%
less-382	23,822	382	46	14,720.8	490	4,906.4	427	66.7%
make-3.76.1	27,304	191	61	14,681.9	695	5,248.0	549	64.3%
wget-1.9	35,018	434	13	6,717.5	544	4,383.4	552	34.7%
screen-4.0.2	44,734	589	77	310,788.0	2,228	66,920.6	1,875	78.5%
bash-2.05a	105,174	959	4	1,637.6	272	1,492.4	265	8.9%

- Some programs contain large recursive call cycles. One common belief for C programs is that it does not largely use recursion in practice. However, our finding from the benchmark programs is that some programs extensively use recursion and large recursive cycles unexpectedly exist in a number of real C programs. For example, from Table 1, note that program `less`, `make`, and `screen` have recursive cycles (scc) that contain more than 40 procedures.
- $\mathsf{Airac}_{\mathsf{AccLoc}}$ is extremely inefficient for those programs. For other programs that have small (or no) recursive cycles, the analysis with access-based localization is quite efficient. For example, analyzing `bash` (the largest one in our benchmark) takes 1,637s. However, analyzing those programs that have large recursive cycles takes much more time: `less` and `make` take more than 10,000s and `screen` takes more than 310,000s to finish the analysis, even though they are not the largest programs.
- $\mathsf{Airac}_{\mathsf{Bypass}}$ is especially effective for those programs. For programs `less`, `make`, and `screen` that contain large recursive cycles, our technique reduces the analysis time by 66.7%, 64.3%, and 78.5%, respectively.
- $\mathsf{Airac}_{\mathsf{Bypass}}$ is also noticeably effective for other programs. For programs, which have small cycles (consisting of less than 20 procedures), $\mathsf{Airac}_{\mathsf{Bypass}}$ saved 8.9%–44.1% of the analysis time of $\mathsf{Airac}_{\mathsf{AccLoc}}$. For example, in analyzing `parser`, $\mathsf{Airac}_{\mathsf{AccLoc}}$ took 572 seconds but $\mathsf{Airac}_{\mathsf{Bypass}}$ took 319 seconds.

Our technique is also likely to reduce peak memory cost. Because our technique localizes memory states more aggressively than the access-based localization, the peak memory consumption must be reduced. However, in the experiments,

memory cost for analyzing smaller programs (`gzip, parser, bc, twolf, tar`) slightly increased. This is because $\mathsf{Airac}_{\mathsf{Bypass}}$ additionally keeps bypassing information on memory. But, for larger programs (`less, make, wget, screen, bash`), the results show that our technique reduces memory costs. For example, $\mathsf{Airac}_{\mathsf{AccLoc}}$ required 2,228 MB in analyzing `screen` but $\mathsf{Airac}_{\mathsf{Bypass}}$ required 1,875 MB.

$\mathsf{Airac}_{\mathsf{Bypass}}$ is at least as precise as $\mathsf{Airac}_{\mathsf{AccLoc}}$. In principle, more aggressive localization improves our analysis because unnecessary values are not passed to procedures and hence avoids needless widening operations. In experiments (similar to one performed in [13,15], the precision of $\mathsf{Airac}_{\mathsf{Bypass}}$ was the same as $\mathsf{Airac}_{\mathsf{AccLoc}}$.

5 Related Work

In static analysis, localization has been widely used for reducing analysis cost [23,22,3,18,19,11,7,12,14], but previous localization methods have a common limitation as described in this paper. Previous localization schemes are classified into reachability-based and access-based. For example, in shape analysis, called procedures are only passed with reachable parts of the heap, which improves the scalability of interprocedural shape analysis [18,19,11,3,23,22]. Similar reachability-based techniques, which removes unreachable bindings, are also popular in higher-order flow analyses [6,7,12]. Access-based localization [14] refines reachability-based approach so that reachable but non-accessed memory locations are additionally removed. The technique was successfully applied to interval-domain-based global static analysis [14]. These localization methods have a common limitation in handling procedure calls. The inefficient aspect, however, has not been well addressed in the literature. We believe the reason is two-folds: (1) because localization itself greatly improves global analysis performance, such 'small' inefficiency is often neglected; (2) the inefficiency only comes to the fore when we analyze programs that have complex procedural features such as large recursive call cycles. In this paper, we show that the problem is one key reason for why localization sometimes does not have satisfactory performance in practice, and propose a solution that extends the access-based localization technique.

Our technique can be considered as a lightweight sparse analysis. While traditional flow-sensitive analysis propagates information along control flow paths, sparse analysis [20,21,16] uses def-use chains and directly propagate data from definition point to its use points, by which unnecessary computational cost is reduced. Our technique is similar to sparse analysis in that we sometimes bypass data, not propagating them along usual control flow paths. Moreover, the concept of delivery points in section 3.3 is similar to ϕ-functions of SSA-based sparse analysis [4,5] in that both reduces the number of additional join operations. However, we do not require def-use chains to be computed in both analysis and computing delivery points, which is the main challenge of sparse analysis in the presence of pointers [4,5].

6 Conclusion

We presented a new technique to mitigate a performance problem of access-based localization technique. Our technique enables access-based localization to efficiently handle complex procedure calls such as recursive cycles. Our technique is general in that it is applicable to any analysis problems that use access-based localization. We proved the effectiveness of our technique by experiments with a realistic global C static analyzer on a variety of open-source benchmarks.

Acknowledgement. We thank Lucas Brutschy for his helpful comments and suggestions on the early development of the idea. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2011-0000971) and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2011.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation, pp. 196–207 (2003)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 238–252 (1977)
3. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Proceedings of the International Symposium on Static Analysis, pp. 240–260 (2006)
4. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 226–238 (2009)
5. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the Symposium on Code Generation and Optimization (2011)
6. Harrison III, W.L.: The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urabana-Champaign (February1989)
7. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 329–341 (1998)
8. Jhee, Y., Jin, M., Jung, Y., Kim, D., Kong, S., Lee, H., Oh, H., Park, D., Yi, K.: Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco (January 2008), ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf
9. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In: Proceedings of the International Symposium on Static Analysis, pp. 203–217 (2005)

10. Jung, Y., Yi, K.: Practical memory leak detector based on parameterized procedural summaries. In: Proceedings of the International Symposium on Memory Management, pp. 131–140 (2008)
11. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient context-sensitive shape analysis with graph based heap models. In: Proceedings of the International Conference on Compiler Construction, pp. 245–259 (2008)
12. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: Proceedings of the ACM SIGPLAN-SIGACT International Conference on Functional Programming, pp. 13–25 (2006)
13. Oh, H.: Large Spurious Cycle in Global Static Analyses and its Algorithmic Mitigation. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 14–29. Springer, Heidelberg (2009)
14. Oh, H., Brutschy, L., Yi, K.: Access Analysis-based Tight Localization of Abstract Memories. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 356–370. Springer, Heidelberg (2011)
15. Oh, H., Yi, K.: An algorithmic mitigation of large spurious interprocedural cycles in static analysis. Software: Practice and Experience 40(8), 585–603 (2010)
16. Reif, J.H., Lewis, H.R.: Symbolic evaluation and the global value graph. In: Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 104–118 (1977)
17. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61 (1995)
18. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 296–309 (2005)
19. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Proceedings of the International Symposium on Static Analysis, pp. 284–302 (2005)
20. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 12–27. ACM, New York (1988)
21. Wegman, M.N., Kenneth Zadeck, F.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 181–210 (1991)
22. 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)
23. Yang, H., Lee, O., Calcagno, C., Distefano, D., O'Hearn, P.: On scalable shape analysis. Technical Memorandum RR-07-10, Queen Mary University of London, Department of Computer Science (November 2007)

A Deductive Database with Datalog and SQL Query Languages

Fernando Sáenz-Pérez[1], Rafael Caballero[2], and Yolanda García-Ruiz[2,⋆]

Grupo de Programación Declarativa (GPD)
[1]Dept. Ingeniería del Software e Inteligencia Artificial
[2]Dept. Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Spain

Abstract. This paper introduces Datalog Educational System (DES), a deductive database which supports both Datalog and SQL as query languages. Since its inception, this system is targeted to educational purposes rather to develop an efficient, competitive system with respect to other existing systems. As distinguishing features, it is free, open-source, multiplatform, interactive, portable, GUI-enabled, implemented following ISO-Prolog and supports extensions to pure Datalog in the form of stratified negation, strong constraints, types, metapredicates, and duplicates. Also, test case generation for SQL views and declarative debugging for Datalog programs and SQL views are supported. SQL statements, following ISO standard, are compiled to Datalog programs and solved by its inference engine. Nonetheless, ODBC connections are also supported, which enables access to external DBMSs and benefit from their solving performance, persistency and scalability.

Keywords: Deductive Databases, Relational Databases, Datalog, SQL, DES.

1 Introduction

We have witnessed recently the advent of new interest on deductive databases and emerging companies promoting deductive technologies. Datalog, as a deductive query language, has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [5], semantic web [7], social networks [16], policy languages [2], and even for optimization [9]. In addition, current companies as LogicBlox, Exeura, Semmle, and Lixto embody Datalog-based deductive database technologies in the solutions they develop.

This paper presents the Datalog Educational System (DES), which born from the need to teach deductive concepts to postgraduate students. As by that time there was no open-source, free, multiplatform, and interactive system, we decided to start this project. It was first released in 2004 and since then, many releases have been published including features as:

⋆ These authors have been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502).

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 66–73, 2011.

- Tabling-based deductive engine implementing stratified negation.
- Datalog and SQL query languages sharing the same database.
- Nulls and outer join operations.
- Duplicates and duplicate elimination.
- Text-based interactive system with commands for changing and examining its state, logging, batch execution, and many more.
- Source-level tracers and declarative debuggers for both Datalog and SQL.
- Strong constraints including types, primary and foreign keys, functional dependencies, and user-defined constraints.
- ODBC connections to seamlessly access external databases.

DES has been developed to be used via an interactive command shell. Nonetheless, more appealing environments are available. On the one hand, DES has been plugged to the multi-platform, Java-based IDE ACIDE [17]. It features syntax colouring, project management, interactive console with edition and history, configurable buttons for commands, and more. On the other hand, Markus Triska contributed with an Emacs environment.

The system is implemented on top of Prolog and it can be run from a state-of-the-art Prolog interpreter (currently, last versions of Ciao, GNU Prolog, SWI-Prolog and SICStus Prolog) on any OS supported by such Prolog interpreter (i.e., almost any HW/SW platform). Portable executables (i.e., they do not need installation and can be run from any directory they are stored) has been also provided for Windows, Linux, and Mac OS X.

Datalog as supported by DES mainly follows Prolog ISO standard [10], whilst SQL follows SQL:2008 ISO standard [11]. DES provides several metapredicates as well, some of them are included to add support for SQL operations:

- *Negation.* `not(Goal)` computes the negation of *`Goal`* by means of negation as failure (closed world assumption (CWA) [20]). *`Goal`* is located at a higher strata than the predicate it occurs [20]).
- *Aggregates.* `group_by(Relation,Grouping_Variables,Condition)` creates groups from *`Relation`* w.r.t. *`Grouping_Variables`* and compute aggregate data with *`Condition`*, which can include expressions with aggregate functions such as `sum(Variable)`, which returns the running sum of *`Relation`* w.r.t. *`Variable`*. If no grouping is needed or it is left to be done automatically, aggregate predicates are also available, as `sum(Relation,Variable,Result)`.
- *Duplicate elimination.* Metapredicate `distinct(Relation)` computes distinct tuples of *`Relation`* when duplicates are enabled (with the command `/duplicates on`). Also, `distinct(Projecting_Variables,Relation)` performs the same but finding distinct tuples for its first argument, which is a subset of variables of the second.
- *Outer join operations.* Nulls and null-related operations coming from the database community are allowed, as, e.g.: `lj(L,R,C)`, which computes the left outer join of relations *`L`* and *`R`* w.r.t. the join condition *`C`* [18].

As a running example, we consider a flight database, which can be defined alternatively from either SQL or Datalog:

```
% SQL:
CREATE TABLE flight(origin STRING,destination STRING,duration INT);
INSERT INTO flight VALUES ('Madrid','Paris',90);
INSERT INTO flight VALUES ('Paris','Oslo',100);
INSERT INTO flight VALUES ('Madrid','London',110);
% Datalog:
:-type(flight(origin:string,destination:string,duration:int)).
flight('Madrid','Paris',90).
flight('Paris','Oslo',100).
flight('Madrid','London',110).
```

Each SQL statement above can be interactively introduced at the system prompt or stored in a file and processed with the command `/process` *`FileName`*. The Datalog program can be also stored in a file and consulted with the command `/consult` *`FileName`*. For inserting tuples (or rules, in general) in Datalog, the command `/assert` *`Rule`* is provided. Whilst types are mandatory in table definition, in Datalog they are optionally declared with the assertion `:-type(`*`Relation`*`,[`*`ColumnType`*`])`, where its second argument is a list of column names and its types (*`ColumnName`*`:`*`Type`*). Once one of the programs above has been consulted, queries can be submitted from the system prompt, as:

```
DES> SELECT destination FROM flight WHERE origin='Madrid'
answer(flight.destination) -> { answer('London'), answer('Paris') }
DES> flight('Madrid',Destination,Duration)
{ flight('Madrid','London',110), flight('Madrid','Paris',90) }
```

However, while the first query returns the tuples of `flight` projected by the argument `destination`, the second does not. To get a similar output relation, *temporary* Datalog views are provided, which allow defining both the projection of columns and renaming of relations:

```
DES> dest(Destination) :- flight('Madrid',Destination,Duration)
{ dest('London'), dest('Paris') }
```

For the consulted program, a predicate dependency graph (PDG) [20] is built, which relates each predicate in the program with all the predicates used to compute its meaning. From this graph, a stratification [20] is computed, if it exists, so that negation and aggregate predicates are not involved in cycles. For solving a query, a subgraph restricted to the predicates occurring in the query is computed, so that only the meaning of relevant predicates are computed, following a top-down driven approach rather than a bottom-up. Even when a given PDG is non-stratifiable, it is possible that the subgraph for a given query could be as long as this subgraph does not involve such offending cycles.

2 System Architecture

Figure 1 barely depicts the system architecture of DES. Datalog programs are stored in an in-memory Prolog database. Datalog queries are solved by the deductive engine relying on a cache to store results from fixpoint computations.

These are computed by using a tabling technique [6], which finds its roots in dynamic programming. The data structure holding these results is called extension table (ET). Displaying the results for a Datalog query amounts to inspect ET for entries matching the query after its solving. In turn, SQL views and tables are stored in two alternative repositories.

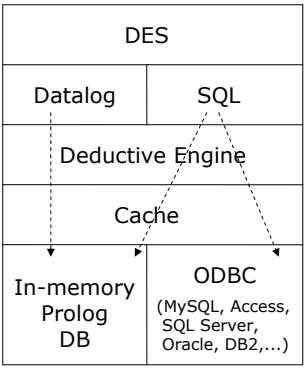


Fig. 1. System Architecture

First alternative for dealing with SQL statements is to use the very same Prolog DB. For this, views are translated into Datalog programs and tables into predicates consisting only of facts. SQL row-returning statements are compiled to and executed as Datalog programs (basics can be found in [20]), and relational metadata for DDL statements are kept. For solving such a query, it is compiled to a Datalog program including the relation `answer`$/n$ with as many arguments as expected from the SQL statement. Then, this program is stored in the Prolog DB, and the Datalog query `answer`$(X_1, \ldots, X_n)$, where $X_i : i \in \{1, \ldots, n\}$ are n fresh variables, is submitted. Results are cached in ET and displayed eventually from this table. After its execution, this Datalog program is removed. On the contrary, if a data definition statement for a view is submitted, its translated program and metadata do persist. This allows Datalog programs to seamlessly use views created at the SQL side (also tables since predicates are used to implement them). The other way round is also possible if types are declared for predicates (further work may include an automatic type assertion via type inferencing). In order to maintain consistency, the cache is cleared whenever the database is updated via asserting Datalog rules, creating SQL views or modifying base tables via `INSERT`, `DELETE` or `UPDATE` SQL statements.

Second alternative is to use the ODBC bridge to access external databases therefore taking advantage from their solving performance, persistency and scalability. Submitting a `CREATE VIEW` SQL statement amounts to forward it to the external database through the ODBC connection. This statement is processed by the external database and operation success is returned to DES, which does not use the Prolog DB for SQL statements anymore. A SQL row-returning statement is also submitted through the bridge, which returns result tuples that are cached by DES. Datalog programs and queries can refer to SQL data because the bridge provides a view to external data sources as Datalog predicates. However, in this second alternative, the other way round is not possible yet, as the external data engine is not aware of the deductive data.

3 Source-to-Source Program Transformations

Asserting a Datalog rule involves several steps. First, parsing builds a syntactic tree for valid rules. If errors are found, an exception is raised with error location and source data. Otherwise, a preprocessing stage is performed, consisting of:

- Simplify successive applications of `not(`*`Goal`*`)` to avoid more strata than strictly needed. As well, applications of this predicate to comparison built-ins (as `=`, `<`, ...) are translated to the complemented versions (`\=`, `>=`, ..., resp.) Finally, a compound goal is also allowed and it is defined as the body of a rule for a brand new predicate where its arguments are the relevant variables in *`Goal`*. The single argument of `not` is replaced by a straight call to this predicate (e.g., the goal `not((p(X),q(X)))` is translated into `not('$p1'(X))` and the rule `'$p1'(X):-p(X),q(X)` is added to the program).
- Aggregate predicates can include compound goals as aggregate relations. These relations are computed before the aggregation itself (sum, average, etc.) following a similar, stratified approach as for computing negation. Therefore, compound goals are also translated as for negation.
- Since disjunctive bodies are allowed, a rule containing a disjunction is transformed to as many rules as needed (e.g., `p(X):-q(X);r(X)` is translated into `p(X):-'$p1'(X)` and the rules `'$p1'(X):-q(X)` and `'$p1'(X):-r(X)` are added to the program).
- Program simplification can be enabled with the command `/simplify on`, which amounts to remove true goals, unify variables, simplify Boolean conditions, and evaluate arithmetic expressions.
- Program transformation for safe rules can also be enabled with the command `/safe on`, which reorder goals in a rule if this rule is unsafe (Section 4).
- Finally, outer joins are translated, first, in order to be solved without resorting to metapredicates as described in [18] and, second, so that relations to be joined are not compound by adding as many predicates as needed (e.g., `lj(rj(p(X),q(Y),X>Y),r(Z),Z>=X)` would be translated into `lj('$p1'(X,Y)),r(Z),Z>=X)` and rule `'$p1'(X,Y):-rj(p(X),q(Y),X>Y)` added to the database).

4 Compile-Time Analyses

DES conducts compile-time analysis to detect unsafe rules. Classic safety [20,22] refers to built-in predicates that can be source of infinite relations, in contrast to user-defined predicates, which are always finite. For instance, the rule `less(X,Y):-X<Y` is unsafe since the built-in `<` can be source of infinite data (its meaning must include any pair such that its first argument is less than its second one). Negation requires its argument to have no unsafe variables, i.e., those which are not bound by a former data provider (as a call to a user-defined predicate). Built-in `X is` *`Expr`* evaluates the arithmetical expression *`Expr`*, so that *`Expr`* is also demanded to be ground and, thus, all its variables must be safe.

Another source of unsafety, departing from the classical notion, resides in metapredicates as `distinct`/2 and aggregates. A set variable is any variable occurring in a metapredicate such that it is not bound by the metapredicate. For instance, `Y` in the goal `distinct([X],t(X,Y))` is a set variable, as well as in `group_by(t(X,Y),[X],C=count)`. Because computing a goal follows SLD order, if a set variable is used after the metapredicate, as in `distinct([X],t(X,Y)),`

`p(Y)`, then this is an unsafe goal as in the call to `distinct`, `Y` is not bound, and all tuples in `t/2` are considered for computing its outcome. Swapping both subgoals yields a safe goal. So data providers for set variables are only allowed before their use in such metapredicates.

Along program transformation, unsafe rules can be automatically generated, as in the translations of outer joins. However, they are safe because of their use: unsafe arguments of such rules are always given as input in goals. So, mode information for predicates is handled throughout program compilations to detect truly unsafe rules, avoiding to raise warnings about system generated rules.

The analysis allows deciding whether a rule is safe and, if so, it is transformed by reordering the goals in order to make it computable. An error is raised when a rule or query is actually unsafe (e.g., the rule `p(X):-X<Y` is unsafe because of `Y`), whereas a warning is issued if the rule *might* be safely computed (e.g., `less(1,2)` can be safely computed since its arguments are ground).

5 Strong Constraints

Consistency constraints over data are known as strong integrity constraints in the deductive database area. Examples of such integrity constraints in the relational field are primary keys and foreign keys, to name a few. As well, constraints in deductive systems as DLV [14] or XSB [19] implementing stable model [8] and well-founded model semantics [21], respectively, are otherwise understood as model filters. In these cases, since a database can have several models, only those fulfilling constraints are included in the answer, therefore discarding *unfeasible* models from the answer. In DES, instead, we focus on integrity constraints as understood in the relational field in order to provide a means to detect inconsistent data with respect to user requirements, including types, primary and foreign keys, functional dependencies, and user-defined constraints.

As an example of constraint, in addition to the type constraint in Section 1, let's consider `:-pk(flight,[origin,destination])`, which defines the column pair in the list to be a primary key for `flight`. Also, assuming:

```
connected(O,D,T) :- flight(O,D,T).
connected(O,D,T) :- flight(O,A,T1),connected(A,D,T2),T is T1+T2.
```

Then, `:-group_by(connected(O,D,T),[O,D],S=sum(T)),S>=300` is a user constraint limiting the duration from an origin to a destination to be less than 300 minutes. Notice that, as usual in the deductive field and contrary to the norm in SQL, an integrity constraint specifies *unfeasible* values rather than feasible.

6 Tracing and Debugging

In contrast to imperative programming languages, deductive and relational database query languages feature solving procedures which are far from the query languages itself. Whilst one can trace an imperative program by following each

statement as it is executed, along with the program state, this is not feasible in declarative (high abstraction) languages as Datalog and SQL.

Similarly, SQL represents a true declarative language which is even farthest from its computation procedure than Prolog. Indeed, the execution plan for a query include transformations considering data statistics to enhance performance. These query plans are composed of primitive relational operations (such as Cartesian product) and specialized operations for which efficient algorithms have been developed, containing in general references to index usage.

Therefore, instead of following a more imperative approach to tracing, here we focus on a (naïve) declarative approach which only take into account the outcomes at some program points. This way, the user can inspect each point and decide whether its outcome is correct or not. This approach will allow examining the syntactical graph of a query, which possibly depends on other views or predicates (SQL or Datalog, resp.) In the case of Datalog queries, this graph contains the nodes and edges in the dependency graph restricted to the query, ignoring other nodes which do not take part in its computation. In the case of SQL, the graph shows the dependencies between a view and its data sources (in the `FROM` clause). Available commands for tracing Datalog and SQL are `/trace_datalog` *`Goal`* and `/trace_sql` *`View`*, respectively.

Algorithmic debugging is also applied to both Datalog and SQL, following [3] and [4], respectively. Similar to how tracing traverses the dependency graph, the debugger in addition prunes paths in the graph by asking the user about validity of its nodes. Available commands for enabling this kind of debugging are `/debug_datalog` *`Goal`* and `/debug_sql` *`View`*.

7 Conclusions

This paper has presented DES, a deductive system used in many universities (`http://des.sourceforge.net/des_facts`) for which its downloading statistics (`http://des.sourceforge.net/statistics`) reveal it as a live project (a new release is expected every two or three months). Statistical numbers show a notable increasing number of downloads, amounting up to more than 1,500 downloads a month, more than 35,000 downloads since 2004.

Features that, as a whole, distinguish DES from other existing systems as DLV [14], XSB [19], bddbddb [13], LDL++ [1], ConceptBase [12], and .QL [15] include null support and outer join operations, duplicates, strong constraints, full-fledged arithmetic, multi-platform, interactiveness, multi-language support, freeness, and open-sourcing, among others. In particular, no one supports outer join operations and full support for duplicates (not only for base relations as LDL++ but also for any rule).

References

1. Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The Deductive Database System LDL++. TPLP 3(1), 61–94 (2003)

2. Becker, M., Fournet, C., Gordon, A.: Design and Semantics of a Decentralized Authorization Language. In: CSF 2007: Proceedings of the 20th IEEE Computer Security Foundations Symposium, pp. 3–15. IEEE, Washington, DC, USA (2007)
3. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Theoretical Framework for the Declarative Debugging of Datalog Programs. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008)
4. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Algorithmic Debugging of SQL Views. In: Ershov Informatics Conference (PSI 2011). Springer, Heidelberg (2011)
5. Calì, A., Gottlob, G., Lukasiewicz, T.: Datalog±: a unified approach to ontologies and integrity constraints. In: ICDT 2009: Proceedings of the 12th International Conference on Database Theory, pp. 14–30. ACM, New York (2009)
6. Dietrich, S.W.: Extension tables: Memo relations in logic programming. In: IEEE Symp. on Logic Programming, pp. 264–272 (1987)
7. Fikes, R., Hayes, P.J., Horrocks, I.: OWL-QL - a language for deductive query answering on the Semantic Web. J. Web Sem. 2(1), 19–29 (2004)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080. MIT Press (1988)
9. Greco, S., Trubitsyna, I., Zumpano, E.: NP Datalog: A Logic Language for NP Search and Optimization Queries. In: International Database Engineering and Applications Symposium, pp. 344–353 (2005)
10. ISO/IEC. ISO/IEC 132111-2: Prolog Standard (2000)
11. ISO/IEC. SQL:2008 9075(1-4,9-11,13,14) Standard (2008)
12. Jarke, M., Jeusfeld, M.A., Quix, C. (eds.): ConceptBase V7.1 User Manual. Technical report, RWTH Aachen (April 2008)
13. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Li, C. (ed.) Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 1–12. ACM (2005)
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Tran. on Computational Logic 7(3), 499–562 (2006)
15. Ramalingam, G., Visser, E. (eds.): Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation. ACM (2007)
16. Ronen, R., Shmueli, O.: Evaluating very large Datalog queries on social networks. In: EDBT 2009: Proceedings of the 12th International Conference on Extending Database Technology, pp. 577–587. ACM, New York (2009)
17. Sáenz-Pérez, F.: ACIDE: An Integrated Development Environment Configurable for LaTeX. The PracTeX Journal 3 (2007)
18. Sáenz-Pérez, F.: Outer joins in a deductive database system. In: XI Jornadas sobre Programación y Lenguajes, PROLE, pp. 126–140 (2011)
19. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: SIGMOD 1994: Proc. of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 442–453. ACM, New York (1994)
20. Ullman, J.D.: Database and Knowledge-Base Systems, vol. I (Classical Database Systems) and II (The New Technologies). Computer Science Press (1988)
21. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 619–649 (1991)
22. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann (1997)

Constructing List Homomorphisms from Proofs

Yun-Yan Chi and Shin-Cheng Mu

IIS, Academia Sinica, Taiwan
{jaiyalas,scm}@iis.sinica.edu.tw

Abstract. The well-known third list homomorphism theorem states that if a function h is both an instance of *foldr* and *foldl*, it is a list homomorphism. Plenty of previous works devoted to constructing list homomorphisms, however, overlook the fact that proving h is both a *foldr* and a *foldl* is often the hardest part which, once done, already provides a useful hint about what the resulting list homomorphism could be. In this paper we propose a new approach: to construct a possible candidate of the associative operator and, at the same time, to transform a *proof* that h is both a *foldr* and a *foldl* to a proof that h is a list homomorphism. The effort constructing the proof is thus not wasted, and the resulting program is guaranteed to be correct.

1 Introduction

A function h on lists is called a *list homomorphism* [1] if it satisfies

$$h\,(xs +\!\!+ ys) = h\,xs \odot h\,ys, \tag{1}$$

for some associative operator $(\odot)$. We wish to identify list homomorphisms due to potential chances of parallelisation: to compute h, one may arbitrarily split the input list into $xs +\!\!+ ys$, compute $h\,xs$ and $h\,ys$ in parallel, and combine the results using $(\odot)$.

The well-known *third list-homomorphism theorem* [7] says that a function is a list homomorphism if it can be described as an instance of both *foldr* and *foldl*. That is, there exists $(\odot)$ satisfying (1) if

$$h = \mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e, \tag{2}$$

for some $(\triangleleft)$ and $(\triangleright)$. For example, $\mathit{sum} = \mathit{foldr}\ (+)\ 0 = \mathit{foldl}\ (+)\ 0$ and, indeed, there exists an $(\odot)$ such that $\mathit{sum}\,(xs +\!\!+ ys) = \mathit{sum}\,xs \odot \mathit{sum}\,ys$ — for this simple example, $(\odot)$ happens to be $(+)$ as well. The proof presented by Gibbons [7] showed that (1) can be satisfied by picking $x \odot y = h\ (h^{-1}\,x +\!\!+ h^{-1}\,y)$, where h^{-1} is a *weak inverse* of h, that is, one such that $h^{-1}\,(h\,x) = x$, which always exists if we assume a set-theoretic semantics.

One naturally wonders whether list homomorphisms can be mechanically constructed. Hu et al. [8] proposed to construct list homomorphisms by fusion with existing ones. Geser and Gorlatch [6] applied term rewriting techniques to construct a definition of $(\odot)$ from that of $(\triangleleft)$ and $(\triangleright)$. More recent developments attempt to apply the third list-homomorphism theorem to mechanical construction

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 74–88, 2011.

of list homomorphisms. Morita et al. [10] proposed to automatically construct $(\odot)$ by picking some weak inverse h^{-1} and simplifying $h\ (h^{-1}\, x + h^{-1}\, y)$. For *sum*, one might pick $sum^{-1}\ x = [x]$, and the system simplifies $sum\ (sum^{-1}\, x + sum^{-1}\, y)$ to $x + y$. For the method to work, it is preferred that h^{-1} has a simple, non-recursive definition, such that $h\ (h^{-1}\, x + h^{-1}\, y)$ can be easily simplified. The method may even be generalised to trees [9].

Elegant as the approach is, when put into practice, however, one cannot help feeling that we have been solving the wrong problem. In all but the most simple cases, efforts are needed to prove (2), that the *foldr* and *foldl* definitions of h do define the same function. It occurs often that one of $(\triangleleft)$ or $(\triangleright)$ is picked as the definition of h, while the other is much harder to find. If the two definitions coincide so obviously that a proof is not necessary, like in the case of *sum*, the choice of $(\odot)$ is often equally trivial that a calculation/proof would be merely stating the obvious.

Once we have both $(\triangleleft)$ and $(\triangleright)$, the operator $(\odot)$ can often be constructed in an ad-hoc but effective manner: we may have a fairly good guess of $(\odot)$ by mixing fragments of code of $(\triangleleft)$ and $(\triangleright)$. We are still left with proving (1), but the proof often turns out to be very similar to that of (2). For a number of examples we fail to see the weak inverse approach applicable: we may have $(\odot)$ constructed, but cannot find any simple h^{-1} that "explains" its discovery.

For such problems, one may turn to the approach of Geser and Gorlatch [6]. An inherent disadvantage of term rewriting, however, is the lack of semantic concerns — having produced some $(\odot)$ offers no direct guarantee that it is correct. One would still like to have a proof of (1).

The way to go, we propose, is to transform the *proof* of (2), which we have to provide anyway, to a proof of (1), after assembling a possible definition of $(\odot)$ from that of $(\triangleleft)$ and $(\triangleright)$.

Program Construction: Syntax v.s. Semantics. In program calculation one transforms a problem specification to a program through algebraic manipulation, thereby guaranteeing its correctness. The program and its correctness proof are developed at the same time.

During the process one is often encouraged to think formally, that is, to think in terms of the syntax rather than the semantics. Rather than focusing on the particular problem domain, the development of the program is ideally driven by syntatical guidelines such as achieving symmetry of expressions, matching the expression against certain forms, and exploiting algebraic properties such as a fusion theorem or the associativity of certain operators. The wish is to relieve programmers of the burden of the complexity in the problem domain through syntactical means — just like how we manipulate arithmetic expressions, using well-designed algebraic rules, without thinking what they "mean." As the slogan says, "let the symbols do the work" [5].

One of the main aims of researchers is therefore to develop convenient notations and theorems that apply generically to a wide range of problems. Such a style, however, could unfortunately have an unhealthy effect when taken to the extreme. Plenty of works on program calculation claim to have discovered

generic theorems that, once the problem specification is put into a certain form or shown to satisfy certain properties, can be applied formally and mechanically to construct an algorithm. Swept under the carpet, however, is the fact that molding the problem into the form or proving the required properties could be hard to carry out formally without domain specific, semantic knowledge. If it can be done, the programmer might have got hold of sufficient knowledge of the problem to just write up the program and, if still necessary at all, prove it correct afterwards.

This shall not be taken as a defect of the methodology. Instead, the value of program calculation is to separate the mechanical, routine process from those critical components that require insight into individual problems. Particularly when we face interesting problems, parts of the development have to be "seen" with the help of the programmer's semantical intuition of the problem, which is then formally proved afterwards. We wish, however, that the efforts doing the proof are not wasted.

Constructing list homomorphisms provides plenty of such examples. As we will see in the forthcoming sections, constructing $(\triangleright)$ from $(\triangleleft)$ in a purely formal manner could be rather hard. The programmer might find it much easier to speculate a possible candidate for $(\triangleright)$, which requires insight into the specific problem, and prove it correct afterwards. Once it is done, however, the construction of $(\odot)$ is relatively mechanical. Our wish is that the effort constructing and proving $(\triangleright)$ is not wasted — it can be used to guide the process finding $(\odot)$.

Contributions. While plenty of previous work devoted to the construction of $(\odot)$ from the definitions of $(\triangleleft)$ and $(\triangleright)$, the novelty of our approach lies in expoiting the information in the proof of that *foldr* $(\triangleleft)$ e equals *foldl* $(\triangleright)$ e. The effort proving (2) is thus not wasted. We find that this approach works for a number of problems that cannot be handled by previous approaches.

After giving a brief review of the concepts needed for this paper in Section 2, we demonstrate our method using three examples: the steep list problem, parallel scan, and a program returning the indexes of those elements in a list that satisfies a given predicate, in Section 3, 4, and 5, before we conclude in Section 6.

2 Preliminaries

We assume a set theoretic model for total functional programming, where a function $A \to B$ is a subset of $A \times B$ that is total (every value in A is mapped to something in B) and simple (every value in A is mapped to at most one value in B).

2.1 Folds and List Homomorphisms

As is well-known, given $e :: B$ and $(\triangleleft) :: A \to B \to B$, the following equations have a unique solution for $h :: [A] \to B$:

$$\begin{aligned} h\ [\,] &= e \\ h\ (x : xs) &= x \triangleleft h\ xs, \end{aligned}$$

which is denoted $foldr\ (\triangleleft)\ e$. The *foldr*-fusion law is among the most important theorems one needs to know about *foldr*:

$$f \circ foldr\ (\triangleleft)\ e = foldr\ (\ll)\ (f\ e) \quad \Leftarrow \quad f\ (x \triangleleft z) = x \ll f\ z.$$

Not all functions are *foldr*s. Let $\langle h, k\rangle\ x = (h\ x, k\ x)$. For all function h taking a list as its input, $\langle h, id\rangle$ can always be defined in terms of a *foldr*. It is also often the case that $\langle h, k\rangle$ could be implemented, as a *foldr*, more efficiently than h alone. The technique of finding the right k to tuple with h is called *tupling* and is now a well-known functional programming technique [8].

Symmetrically, $foldl\ (\triangleright)\ e$ is the unique solution for h given the equations:

$$\begin{aligned} h\ [\,] &= e \\ h\ (xs +\!\!+ [x]) &= h\ xs \triangleright x, \end{aligned}$$

where $(\triangleright) :: B \to A \to B$. Like *foldr*, given h, we can often compute it faster in a *foldl* by tupling it with another function.

A function h is called a *list homomorphism* if it satisfies the following equations for some e, f, and $(\odot)$:

$$\begin{aligned} h\,[\,] &= e \\ h\,[x] &= f\ x \\ h\ (xs +\!\!+ ys) &= h\ xs \odot h\ ys. \end{aligned}$$

The equations imply that $(\odot)$ is associative with identity e. If the equations hold, we denote h by $hom\ (\odot)\ f\ e$. For any function h on lists, $\langle h, id\rangle$ is always a list homomorphism [4]. The equations form a terminating definition if xs and ys in the last clause are restricted to non-empty lists. When h is also defined as a *foldr* or a *foldl*, the cases for $h\,[\,]$ and $h\,[x]$ are determined and often omitted in this paper.

2.2 From Duality to Homomorphism

The *second duality theorem* of Bird [2, page 128] states that $foldr\ (\triangleleft)\ e = foldl\ (\triangleright)\ e$ if

$$z \triangleleft e = e \triangleright z \quad \wedge \quad (x \triangleleft y) \triangleright z = x \triangleleft (y \triangleright z). \tag{3}$$

When (3) holds, $(\triangleleft)$ and $(\triangleright)$ are said to *associate with each other*.[1] We present here a quick proof of the theorem. Let $h = foldr\ (\triangleleft)\ e$. To show that $h = foldl\ (\triangleright)\ e$ it is sufficient to prove that $h\ (xs +\!\!+ [z]) = h\ xs \triangleright z$, which, written point-free, is $h \circ (+\!\!+[z]) = (\triangleright\ z) \circ h$. We perform *foldr*-fusion on both sides:

$$\begin{aligned} & h \circ (+\!\!+[z]) \\ = \quad & \{\ \ foldr\text{-fusion, since } (+\!\!+[z]) = foldr\ (:)\ [z] \ \ \} \end{aligned}$$

[1] By Bird [2]. One could argue that (3) is commutivity: $(\triangleright\ z) \circ (x \triangleleft) = (x \triangleleft) \circ (\triangleright\ z)$.

$$
\begin{aligned}
& \mathit{foldr}\,(\triangleleft)\,(h\,[z]) \\
= \quad & \{\ \mathit{foldr}\text{-fusion (backwards), see below}\ \} \\
& (\triangleright\, z) \circ \mathit{foldr}\ (\triangleleft)\ e \\
= \quad & (\triangleright\, z) \circ h.
\end{aligned}
$$

The fusion conditions of the first *foldr*-fusion trivially holds, while those for the second fusion are $h\,[z] = (\triangleright\, z)\ e$ and $(\triangleright\, z)(x \triangleleft y) = x \triangleleft ((\triangleright\, z)\ y)$, which expands to (3).

To show that h is a list homomorphism, we have to construct $(\odot)$ such that $h\,(xs +\!\!+\, ys) = h\,xs \odot h\,ys$. The calculation is very similar to the one above. The equation can be written in point-free style as $h \circ (+\!\!+ ys) = (\odot\, h\, ys) \circ h$. To find out conditions under which the equality holds, we perform *foldr*-fusion on both sides:

$$
\begin{aligned}
& h \circ (+\!\!+ ys) \\
= \quad & \{\ \mathit{foldr}\text{-fusion, since } (+\!\!+ ys) = \mathit{foldr}\,(:)\ ys\ \} \\
& \mathit{foldr}\,(\triangleleft)\,(h\,ys) \\
= \quad & \{\ \mathit{foldr}\text{-fusion (backwards), see below}\ \} \\
& (\odot h\, ys) \circ \mathit{foldr}\ (\triangleleft)\ e \\
= \quad & (\odot h\, ys) \circ h,
\end{aligned}
$$

which follows a pattern similar to the calculation above. The conditions we need for the second fusion are $h\,ys = (\odot\, h\, ys)\ e$ and $(\odot\, h\, ys)\ (x \triangleleft y) = x \triangleleft ((\odot\, h\, ys)\ y)$, that is,

$$h\,ys = e \odot h\,ys \quad \wedge \quad (x \triangleleft y) \odot h\,ys = x \triangleleft (y \odot h\,ys). \tag{4}$$

Since $(\triangleleft)$ is in essence a special case of $(\odot)$, it is not surprising that a proof of (3) can be generalised to a proof of (4). In fact, inspecting the proof of (3) may give us useful hints what $(\odot)$ could be.

Our aim, therefore, is to come up with a definition of $(\odot)$, together with its correctness proof, given e, $(\triangleleft)$, $(\triangleright)$, and a proof of their associativity.

There is no reason to bias on one side, though. With a symmetric development, one can show that h is a list homomorphism if

$$h\,xs = h\,xs \odot e \quad \wedge \quad h\,xs \odot (y \triangleright z) = (h\,xs \odot y) \triangleright z. \tag{5}$$

Both directions will be handy in this paper.

3 The Steep List Problem

A list of numbers is said to be *steep* if it descends (or ascends, if read right-to-left) so rapidly that each number is larger than the sum of the numbers to its right. Formally:

$$
\begin{aligned}
& \mathit{steep} && :: [\mathit{Int}] \to \mathit{Bool} \\
& \mathit{steep}\ [\,] && = \mathit{True} \\
& \mathit{steep}\ (x : xs) && = x > \mathit{sum}\ xs\ \wedge\ \mathit{steep}\ xs.
\end{aligned}
$$

The problem has been used as an introductory example to tupling: definition of *steep* above is a quadratic time program, while the function $steepsum = \langle steep, sum \rangle$ can be computed in linear time in a *foldr* [3].

Can we determinine the steepness of a list in a *foldl* and, thereby, in a list homomorphism? It turns out that *steep* does not carry enough information to be computed in a *foldl* and we have to generalise further. Let *cap xs*, the *capacity* of *xs*, be a (non-inclusive) upper-bound of values we can attach to the right of *xs* and still keep it steep. That is, for all y, $xs +\!\!+ [y]$ is steep if and only if $y < cap\ xs$. For example, $cap\ [9,5,3] = 1$, since $9 \not> 5 + 3 + x$ for $x > 1$; also, $cap\ [9,4,2] = 2$, since $4 \not> 2 + x$ for $x > 2$.

Definition of *cap* is generalised from that of *steep*:

$$\begin{aligned} &cap && :: [Int] \to Int \\ &cap\ [\,] && = \infty \\ &cap\ (x : xs) && = (x - sum\ xs) \downarrow cap\ xs, \end{aligned}$$

where $(\downarrow)$ returns the minimum of its two arguments. Some intuition for the inductive case: for $x : xs +\!\!+ [y]$ to be steep, we need $x > sum\ xs + y$, and thus the "margin" $x - sum\ xs$ is an upper-bound for y. Furthermore, $xs +\!\!+ [y]$ must be steep as well. Therefore, inductively, *cap xs* is another bound for y. By formalising the argument one easily comes up with an inductive proof that $steep\ xs \equiv cap\ xs > 0$.

Since *sum* and *cap* are both *foldr*s, so is $capsum\ xs = (cap\ xs, sum\ xs)$:

$$\begin{aligned} &capsum && :: [Int] \to (Int, Int) \\ &capsum\ [\,] && = (\infty, 0) \\ &capsum\ (x : xs) && = \mathbf{let}\ (c_2, s_2) = capsum\ xs\ \mathbf{in}\ ((x - s_2) \downarrow c_2, x + s_2). \end{aligned}$$

Can *cap* be computed in a *foldl*? As we will see in Section 3.1, it is rather difficult to construct, purely by formal calculation, a *foldl* definition of *cap* from the *foldr* definition (and vice versa). By thinking semantically, one might guess that

$$cap\ (xs +\!\!+ [z]) = (cap\ xs - z) \downarrow z.$$

The rationale is that $cap\ (xs +\!\!+ [z])$ is the minimum of two upper-bounds: firstly, having z on the right-end lowers the upper-bound *cap xs* by z, and secondly, z itself is a upper-bound. We may thus also define *capsum* as a *foldl*:

$$\begin{aligned} &capsum\ [\,] && = (\infty, 0) \\ &capsum\ (xs +\!\!+ [z]) && = \mathbf{let}\ (c_1, s_1) = capsum\ xs\ \mathbf{in}\ ((c_1 - z) \downarrow z, s_1 + z). \end{aligned}$$

Without a proof, however, one cannot be confident that the two definitions of *cap* (and thus *capsum*) do define the same function. Once the proof of their equivalence is done, however, we would be taking an unnecessarily long route if we abandon all the efforts above, start from scratch, and try to construct $(c_1, s_1) \odot (c_2, s_2) = capsum\ (capsum^{-1}\ (c_1, s_1) +\!\!+ capsum^{-1}\ (c_2, s_2))$ — for this example, in fact, we have failed to come up with a simple $capsum^{-1}$.

Instead, it turns out that the homomorphic definition of *capsum* is assembled from parts of its *foldr* and *foldl* definitions:

$$\begin{aligned} capsum\ (xs +\!\!+ ys) = \mathbf{let}\ \{\ &(c_1, s_1) = capsum\ xs;\ (c_2, s_2) = capsum\ ys\ \} \\ \mathbf{in}\ &((c_1 - s_2) \downarrow c_2, s_1 + s_2). \end{aligned}$$

Still, one needs a proof that the definition above does coincide with the *foldr* definition. The proof highly resembles the proof of the associativity. One thus wonders whether it is possible to somehow reuse the definitions and proofs, which will be done in Section 3.2.

3.1 Constructing $(\triangleright)$ Formally

In the opening of this section we have argued, semantically, that $\mathit{capsum} = \mathit{foldr}\ (\triangleleft)\ (\infty, 0) = \mathit{foldl}\ (\triangleright)\ (\infty, 0)$, where

$$x \triangleleft (c_2, s_2) = ((x - s_2) \downarrow c_2, x + s_2),$$
$$(c_1, s_1) \triangleright z = ((c_1 - z) \downarrow z, s_1 + z).$$

Before we proceed with constructing the homomorphic definition of *capsum*, we show in this section that it is hard to construct $(\triangleright)$ from $(\triangleleft)$ in a purely syntactical manner. This echos our observation in the next subsection that the proof of $(x \triangleleft y) \triangleright z = x \triangleleft (y \triangleright z)$, which has to be done if $(\triangleright)$ is not derived, already provides plenty of information necessary to construct $(\odot)$. Finding $(\triangleright)$ is thus where all the hard work is.

Here is an attempt. We assume that $(c, s) \triangleright z = (f_1\ c\ s\ z, f_2\ c\ s\ z)$ for some f_1 and f_2, which we shall try to construct such that $z \triangleleft (\infty, 0) = (\infty, 0) \triangleright z$ and $(x \triangleleft (c, s)) \triangleright z = x \triangleleft ((c, s) \triangleright z)$. We start from the left-hand side of the latter:

$$\begin{aligned}
& (x \triangleleft (c, s)) \triangleright z \\
= & \quad \{\ \text{definition of } (\triangleleft)\ \} \\
& ((x - s) \downarrow c, x + s) \triangleright z \\
= & \quad \{\ \text{definition of } (\triangleright)\ \} \\
& (f_1\ ((x - s) \downarrow c)\ (x + s)\ z, f_2\ ((x - s) \downarrow c)\ (x + s)\ z).
\end{aligned}$$

Now that we are stuck, we expand the right-hand side:

$$\begin{aligned}
& x \triangleleft ((c, s) \triangleright z) \\
= & \quad \{\ \text{definition of } (\triangleright)\ \} \\
& x \triangleleft (f_1\ c\ s\ z, f_2\ c\ s\ z) \\
= & \quad \{\ \text{definition of } (\triangleleft)\ \} \\
& ((x - f_2\ c\ s\ z) \downarrow f_1\ c\ s\ z, x + f_2\ c\ s\ z).
\end{aligned}$$

We shall try to somehow unify the two resulting expressions.

The simplest choice of f_2 such that $f_2\ ((x - s) \downarrow c)\ (x + s)\ z = x + f_2\ c\ s\ z$ would be $f_2\ c\ s\ z = s$, which unfortunately fails the requirement that $z \triangleleft (\infty, 0) = (\infty, 0) \triangleright z$, which expands to

$$(z, z) = (f_1\ \infty\ 0\ z, f_2\ \infty\ 0\ z). \tag{6}$$

It may be the next logical choice to try $f_2\ c\ s\ z = s + z$, which turns out to be correct. Having found f_2, we shall then unify

$$f_1\ ((x - s) \downarrow c)\ (x + s)\ z \quad \text{and} \quad (x - (s + z)) \downarrow f_1\ c\ s\ z.$$

Which is no easy task. The simplest choice is $f_1\ c\ s\ z = c - z$, which unifies the two expressions,

$$\begin{aligned} f_1\ ((x-s)\downarrow c)\ (x+s)\ z &= ((x-s)\downarrow c) - z \\ &= (x-s-z)\downarrow(c-z) = (x-(s+z))\downarrow f_1\ c\ s\ z, \end{aligned}$$

but again turns out to be wrong because it fails the requirement in (6) that $f_1\ \infty\ 0\ z = z$. It takes some creativity to come up with $f_1\ c\ s\ z = (c\downarrow z) - z$, which is best explained by the semantical view in the beginning of this section.

3.2 Computing Capacity by a List Homomorphism

Return to the problem of constructing *capsum* as a list homomorphism. From the discussion in Section 2.2, to prove that $capsum = foldr\ (\triangleleft)\ (\infty, 0) = foldl\ (\triangleright)\ (\infty, 0)$ we need to prove that

$$z \triangleleft (\infty, 0) = (\infty, 0) \triangleright z \quad \wedge \quad (x \triangleleft y) \triangleright z = x \triangleleft (y \triangleright z),$$

where $x \triangleleft (c_2, s_2) = ((x - s_2)\downarrow c_2, x + s_2)$ and $(c_1, s_1) \triangleright z = ((c_1 - z)\downarrow z, s_1 + z)$. While we have have seen in Section 3.1 that it is hard to construct $(\triangleright)$ in a purely syntactical manner, once we have somehow conjectured $(\triangleright)$, the proof of its correctness is routine. To show that $(\triangleleft)$ and $(\triangleright)$ associate, we reason:

$$\begin{aligned} & (x \triangleleft (c, s)) \triangleright z \\ = & \quad \{\ \text{definition of } (\triangleleft)\ \} \\ & ((x-s)\downarrow c, x+s) \triangleright z \\ = & \quad \{\ \text{definition of } (\triangleright)\ \} \\ & ((((x-s)\downarrow c) - z)\downarrow z, x+s+z) \\ = & \quad \{\ (-z) \text{ distributes over } (\downarrow)\ \} \\ & (((x-s-z)\downarrow(c-z))\downarrow z, x+s+z) \\ = & \quad \{\ \text{arithmetics}\ \} \\ & (((x-(s+z))\downarrow((c-z)\downarrow z), x+s+z) \\ = & \quad \{\ \text{definition of } (\triangleleft)\ \} \\ & x \triangleleft ((c-z)\downarrow z, s+z) \\ = & \quad \{\ \text{definition of } (\triangleright)\ \} \\ & x \triangleleft ((c, s) \triangleright z). \end{aligned}$$

The proof will be referred to as the "proof of associativity". The aim now is to construct a definition of $(\odot)$ and generalise the proof above to a proof of

$$(c_2, s_2) = (\infty, 0) \odot (c_2, s_2) \quad \wedge \quad (x \triangleleft y) \odot (c_2, s_2) = x \triangleleft (y \odot (c_2, s_2)),$$

for all (c_1, s_1) in the range of *capsum*.

Since $(\odot)$ is a generalisation of $(\triangleright)$, we start with replacing the occurrences of z in $(\triangleright)$ by metavariables. Our first guess is

$$(c_1, s_1) \odot (c_2, s_2) = ((c_1 - X_1) \downarrow X_2, s_1 + X_3).$$

To prove that $(x \triangleleft y) \odot (c_2, s_2) = x \triangleleft (y \odot (c_2, s_2))$ we copycat the steps in the proof of associativity. Starting from $(x \triangleleft (c, s)) \odot (c_1, s_1)$, we reason:

$$\begin{aligned}
& (x \triangleleft (c, s)) \odot (c_2, s_2) \\
= & \quad \{ \text{ definition of } (\triangleleft) \ \} \\
& ((x - s) \downarrow c, x + s) \odot (c_2, s_2) \\
= & \quad \{ \text{ definition of } (\odot) \ \} \\
& ((((x - s) \downarrow c) - X_1) \downarrow X_2, x + s + X_3) \\
= & \quad \{ \ (-X_1) \text{ distributes over } (\downarrow) \ \} \\
& (((x - s - X_1) \downarrow (c - X_1)) \downarrow X_2, x + s + X_3) \\
= & \quad \{ \text{ arithmetics } \} \\
& (((x - (s + X_1)) \downarrow ((c - X_1) \downarrow X_2), x + s + X_3).
\end{aligned}$$

In the next step we are supposed to fold back the definition of $(\triangleleft)$. To be able to do so, however, $(s + X_1)$ and $(s + X_3)$ have to be the same term and we thus have to unify X_1 and X_3. The last two steps go:

$$\begin{aligned}
& (((x - (s + X_1)) \downarrow ((c - X_1) \downarrow X_2), x + s + X_1) \\
= & \quad \{ \text{ definition of } (\triangleleft) \ \} \\
& x \triangleleft ((c - X_1) \downarrow X_2, s + X_1) \\
= & \quad \{ \text{ definition of } (\odot) \ \} \\
& x \triangleleft ((c, s) \odot (c_2, s_2))
\end{aligned}$$

Thus the definition of $(\odot)$ is now refined to

$$(c_1, s_1) \odot (c_2, s_2) = ((c_1 - X_1) \downarrow X_2, s_1 + X_1),$$

for some X_1 and X_2. Notice that any choice of X_1 and X_2 would allow the proof to go through. In a sense, we have exploited all information from the proof above; it could tell us no more about X_1 and X_2!

To figure out X_1 and X_2 we turn to the base case, where to have to show that $(c_2, s_2) = (\infty, 0) \odot (c_2, s_2)$. Expanding the right-hand side, we get

$$(c_2, s_2) = ((\infty - X_1) \downarrow X_2, 0 + X_1).$$

An obvious choice would be $X_1 = s_2$ and $X_2 = c_2$. We have thus discovered that $(c_1, s_1) \odot (c_2, s_2) = ((c_1 - s_2) \downarrow c_2, s_1 + s_2)$. This $(\odot)$ has got to be correct, because we have the proof already!

As a remark, similar principles can be applied to derive a list-homomorphic solution of the maximum segment sum problem. We will need a four-tuple whose components respectively store the maximum segment sum, prefix sum, suffix sum, and the total sum. The calculation is tedious, but not essentially harder.

4 Parallel Scan

Constructing the list-homomorphic definition of *scanr* is dealt with in Hu et al. [8] and Geser and Gorlatch [6], but notably not in Morita et al [10]. In this section we present our solution. The function $scanr\ (\oplus)\ e$ applies $foldr\ (\oplus)\ e$ to every tail of its input list. It is well known that $scanr\,(\oplus)\,e = foldr\,(\triangleleft)\,[e] = foldl\,(\triangleright)\,[e]$ if $(\oplus)$ is associative with unit e, where

$$x \triangleleft ys = (x \oplus head\ ys) : ys,$$
$$ys \triangleright z = map\ (\oplus z)\ ys +\!\!+ [e].$$

The proof of associativity of $(\triangleleft)$ and $(\triangleright)$ is given below, where ys, being in the range of $scanr\,(\oplus)\,e$, is a non-empty list.

$$\begin{array}{ll} & (x \triangleleft ys) \triangleright z \\ = & \quad \{\ \text{definition of } (\triangleleft)\ \} \\ & ((x \oplus head\ ys) : ys) \triangleright z \\ = & \quad \{\ \text{definition of } (\triangleright)\ \} \\ & map\ (\oplus z)\ ((x \oplus head\ ys) : ys) +\!\!+ [e] \\ = & \quad \{\ \text{definition of } map\ \} \\ & ((x \oplus head\ ys) \oplus z) : map\ (\oplus z)\ ys +\!\!+ [e] \\ = & \quad \{\ \text{associativity of } \oplus\ \} \\ & (x \oplus (head\ ys \oplus z)) : map\ (\oplus z)\ ys +\!\!+ [e] \\ = & \quad \{\ f(head\ xs) = head\ (map\ f\ xs)\ \} \\ & (x \oplus head\ (map\ (\oplus z)\ ys)) : map\ (\oplus z)\ ys +\!\!+ [e] \\ = & \quad \{\ head\ xs = head\ (xs +\!\!+ ys) \text{ if } xs \text{ non-empty}\ \} \\ & (x \oplus head\ (map\ (\oplus z)\ ys +\!\!+ [e])) : map\ (\oplus z)\ ys +\!\!+ [e] \\ = & \quad \{\ \text{definition of } (\triangleleft)\ \} \\ & x \triangleleft (map\ (\oplus z)\ ys +\!\!+ [e]) \\ = & \quad \{\ \text{definition of } (\triangleright)\ \} \\ & x \triangleleft (ys \triangleright z). \end{array}$$

We aim to construct $(\odot)$ and generalise the proof above to a proof of

$$zs = [e] \odot zs \quad \wedge \quad (x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs),$$

where zs is also an non-empty list. One possible candidate of $(\odot)$ is obtained by replacing the sole occurrence of z in $(\triangleright)$ by a metavariable:

$$ys \odot zs = map\ (\oplus X_1)\ ys +\!\!+ [e].$$

We may proceed with it and show that it indeed satisfies every step in the proof, that is, for such a choice it is true that $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$, whatever X_1 is.

This candidate fails, however, when we consider the base case $zs = [e] \odot zs$, which expands to $zs = X_1 : [e]$ and restricts zs to lists having exactly two elements. Apparently our $(\odot)$ is not general enough.

To allow zs to be of arbitrary length, one possibility is to generalise $[e]$ to X_2 — an instance of the common technique to generalise occurrences of constants to metavariables. The proof goes:

$$
\begin{array}{ll}
 & (x \lhd ys) \odot zs \\
= & \quad \{ \text{ definition of } (\lhd) \ \} \\
 & ((x \oplus head\ ys) : ys) \odot zs \\
= & \quad \{ \text{ conjecture: } ys \odot zs = map\ (\oplus X_1)\ ys +\!\!+ X_2 \ \} \\
 & map\ (\oplus X_1)\ ((x \oplus head\ ys) : ys) +\!\!+ X_2 \\
= & \quad \{ \text{ definition of } map \ \} \\
 & ((x \oplus head\ ys) \oplus X_1) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \quad \{ \text{ associativity of } (\oplus) \ \} \\
 & (x \oplus (head\ ys \oplus X_1)) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \quad \{ \ f\ (head\ xs) = head\ (map\ f\ xs) \ \} \\
 & (x \oplus head\ (map\ (\oplus X_1)\ ys)) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \quad \{ \ head\ xs = head\ (xs +\!\!+ ys), \text{ if } xs \text{ non-empty} \ \} \\
 & (x \oplus head\ (map\ (\oplus X_1)\ ys) +\!\!+ X_2) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \quad \{ \text{ definition of } (\lhd) \ \} \\
 & x \lhd (map\ (\oplus X_1)\ ys +\!\!+ X_2) \\
= & \quad \{ \text{ definition of } (\odot) \ \} \\
 & x \lhd (ys \odot zs).
\end{array}
$$

Thus another possible choice is $ys \odot zs = map\ (\oplus X_1)\ ys +\!\!+ X_2$, which also allows the proof of $(x \lhd ys) \odot zs = x \lhd (ys \odot zs)$ to go through. The base case $zs = [e] \odot zs$, this time, expands to

$$zs = map\ (\oplus X_1)\ [e] +\!\!+ X_2 = X_1 : X_2.$$

We may pick $X_1 = head\ zs$ and $X_2 = tail\ zs$, and have thus discovered $(\odot)$,

$$ys \odot zs = map\ (\oplus\ head\ zs)\ ys +\!\!+ tail\ zs.$$

5 Reasoning with Conditionals

Our last example is chosen to demonstrate calculation involving conditional expressions, and the symmetric property that h is a list homomorphism if (5) holds, which is repeated here,

$$h\ xs = h\ xs \odot e \quad \wedge \quad h\ xs \odot (y \rhd z) = (h\ xs \odot y) \rhd z.$$

Given a predicate $p :: a \to Bool$, the function $pos\ p :: [a] \to [Int]$ returns the indexes of elements in the input list that satisfy p. For example, $pos\ (\geq 5)\ [6,8,4,2,0,10] = [0,1,5]$. For a definition,

$$\begin{array}{lll} pos & :: (a \to Bool) \to [a] \to [Int] \\ pos\ p\ [\,] & = [\,] \\ pos\ p\ (x : xs) & = \textbf{let}\ ys = pos\ p\ xs \\ & \quad \textbf{in if}\ p\ x\ \textbf{then}\ 0 : map\ (1+)\ ys\ \textbf{else}\ map\ (1+)\ ys. \end{array}$$

To compute pos in a $foldl$, we have to tuple it with $length$. Let $poslen = \langle pos, length \rangle$, we claim that $poslen = foldr\ (\triangleleft)\ ([\,],0) = foldl\ (\triangleright)\ ([\,],0)$, where

$$\begin{array}{l} x \triangleleft (ys, n) = (\textbf{if}\ p\ x\ \textbf{then}\ 0 : map\ (1+)\ ys\ \textbf{else}\ map\ (1+)\ ys, 1+n), \\ (ys, n) \triangleright z = (\textbf{if}\ p\ z\ \textbf{then}\ ys +\!\!+ [n]\ \textbf{else}\ ys, n+1). \end{array}$$

When we construct a proof of (4) in Section 3.2, the step where we fold back the definition of $(\triangleleft)$, due to repeated occurrences of s_2, is the step that triggered unification of metavariables. For $poslen$, we could construct $(\odot)$ and a proof of (4), but more guesswork is involved. Since $(\triangleright)$ for $poslen$ shares one more variable, n, in both components of the pair, it could be, and indeed is, easier to try the other direction — to generalise the proof of associativity to a proof of (5). That way we will be folding $(\triangleright)$ instead of $(\triangleleft)$, which possibly allows more unification to happen.

That $z \triangleleft ([\,],0) = ([\,],0) \triangleright z$ can be easily verified. The proof that $(\triangleleft)$ and $(\triangleright)$ associate is given below. To adapt to (5), the proof starts from a different side. For brevity we use the Bird-Meertens notation denoting $map\ f$ by f^*, and denote $\textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2$ by $\langle\!\langle p \to e_1, e_2 \rangle\!\rangle$.

$$\begin{array}{ll} & x \triangleleft ((ys, n) \triangleright z) \\ = & \quad \{\ \text{definition of } (\triangleright)\ \} \\ & x \triangleleft (\langle\!\langle p\,z \to ys +\!\!+ [n],\ ys \rangle\!\rangle, n+1) \\ = & \quad \{\ \text{definition of } (\triangleleft)\ \} \\ & (\langle\!\langle p\,x \to 0 : (1+)^*\ \langle\!\langle p\,z \to ys +\!\!+ [n],\ ys \rangle\!\rangle, \\ & \qquad (1+)^*\ \langle\!\langle p\,z \to ys +\!\!+ [n],\ ys \rangle\!\rangle\!\rangle, \\ & \quad 1+n+1) \\ = & \quad \{\ \text{transposition of nested-}\textbf{if}\ \} \\ & (\langle\!\langle p\,z \to \langle\!\langle p\,x \to (0:) \circ (1+)^*,\ (1+)^* \rangle\!\rangle\ (ys +\!\!+ [n]), \\ & \qquad \langle\!\langle p\,x \to (0:) \circ (1+)^*,\ (1+)^* \rangle\!\rangle\ ys \rangle\!\rangle, \\ & \quad 1+n+1) \\ = & \quad \{\ \langle\!\langle b \to f_1\ a, f_2\ a \rangle\!\rangle = \langle\!\langle b \to f_1, f_2 \rangle\!\rangle\ a\ \} \\ & (\langle\!\langle p\,z \to \langle\!\langle p\,x \to 0 : ((1+)^*\ ys),\ (1+)^*\ ys \rangle\!\rangle +\!\!+ [1+n], \\ & \qquad \langle\!\langle p\,x \to 0 : ((1+)^*\ ys),\ (1+)^*\ ys \rangle\!\rangle\!\rangle, \\ & \quad 1+n+1) \end{array}$$

$$
\begin{aligned}
&= \quad \{ \text{ definition of } (\triangleright) \ \} \\
&\quad (\langle\!\langle p\, x \to 0 : (1+)^*\ ys,\ (1+)^*\ ys \rangle\!\rangle, 1+n) \triangleright z \\
&= \quad \{ \text{ definition of } (\triangleleft) \ \} \\
&\quad (x \triangleleft (ys, n)) \triangleright z.
\end{aligned}
$$

The rule "transposition of nested-**if**" states that

$$
\begin{aligned}
&\langle\!\langle p \to \langle\!\langle q \to f_1,\ f_2 \rangle\!\rangle\ x_1,\ \langle\!\langle q \to f_1,\ f_2 \rangle\!\rangle\ x_2 \rangle\!\rangle \\
&= \langle\!\langle q \to\ f_1\ \langle\!\langle p \to x_1,\ x_2 \rangle\!\rangle,\ f_2\ \langle\!\langle p \to x_1,\ x_2 \rangle\!\rangle \rangle\!\rangle.
\end{aligned}
$$

The next step is to construct a definition of $(\odot)$ together with a proof of $h\ xs \odot (y \triangleright z) = (h\ xs \odot y) \triangleright z$ from the proof of associativity. A possible candidate of $(\odot)$ is generalised from the definition of $(\triangleleft)$:

$$
(ys_1, n_1) \odot (ys_2, n_2) = (\langle\!\langle X_1 \to\ X_2 +\!\!+ (X_3+)^*\ ys_2,\ (X_4+)^*\ ys_2 \rangle\!\rangle, X_5 + n).
$$

Applying the lessons learnt in previous sections, we replace subterms involving x to metavariables, and replace constants by metavariables in a way that allows flexibility in length.

Now we try to construct a proof of $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$. We follow the steps of the proof of associativity:

$$
\begin{aligned}
&\quad (ys_1, n_1) \odot ((ys, n) \triangleright z) \\
&= \quad \{ \text{ definition of } (\triangleright) \ \} \\
&\quad (ys_1, n_1) \odot (\langle\!\langle p\, z \to\ ys +\!\!+ [n],\ ys \rangle\!\rangle, n+1) \\
&= \quad \{ \text{ definition of } (\odot) \ \} \\
&\quad (\langle\!\langle X_1 \to X_2 +\!\!+ (X_3+)^*\ \langle\!\langle p\, z \to\ ys +\!\!+ [n],\ ys \rangle\!\rangle, \\
&\qquad\quad (X_4+)^*\ \langle\!\langle p\, z \to\ ys +\!\!+ [n],\ ys \rangle\!\rangle \rangle\!\rangle, \\
&\qquad X_5 + n + 1) \\
&= \quad \{ \text{ transposition of nested-}\textbf{if} \ \} \\
&\quad (\langle\!\langle p\, z \to \langle\!\langle X_1 \to\ (X_2+\!\!+) \circ (X_3+)^*,\ (X_4+)^* \rangle\!\rangle\ (ys +\!\!+ [n]), \\
&\qquad\quad \langle\!\langle X_1 \to\ (X_2+\!\!+) \circ (X_3+)^*,\ (X_4+)^* \rangle\!\rangle\ ys \rangle\!\rangle, \\
&\qquad X_5 + n + 1),
\end{aligned}
$$

Unification happens in the next two steps. In the first step, to move ys into the conditional while leaving n outside, X_3 and X_4 must be unified. In the second step, X_5 and X_3 are unified to allow the definition of $(\triangleright)$ to fold:

$$
\begin{aligned}
&\quad (\langle\!\langle p\, z \to \langle\!\langle X_1 \to\ (X_2+\!\!+) \circ (X_3+)^*,\ (X_4+)^* \rangle\!\rangle\ (ys +\!\!+ [n]), \\
&\qquad\quad \langle\!\langle X_1 \to\ (X_2+\!\!+) \circ (X_3+)^*,\ (X_3+)^* \rangle\!\rangle\ ys \rangle\!\rangle, \\
&\qquad X_3 + n + 1) \\
&= \quad \{ \ \langle\!\langle b \to f_1\ a,\ f_2\ a \rangle\!\rangle = \langle\!\langle b \to f_1, f_2 \rangle\!\rangle\ a \ \} \\
&\quad (\langle\!\langle p\, z \to \langle\!\langle X_1 \to\ X_2 +\!\!+ ((X_3+)^*\ ys),\ (X_3+)^*\ ys \rangle\!\rangle +\!\!+ [X_3 + n],
\end{aligned}
$$

$$
\begin{array}{ll}
 & \langle\!\langle X_1 \to X_2 +\!\!+ ((X_3+)^*\ ys),\ (X_3+)^*\ ys \rangle\!\rangle, \\
 & X_3 + n + 1) \\
= & \quad \{\ \text{definition of } (\triangleright)\ \} \\
 & (\langle\!\langle X_1 \to X_2 +\!\!+ (X_3+)^*\ ys,\ (X_3+)^*\ ys \rangle\!\rangle, X_3 + n) \triangleright z \\
= & \quad \{\ \text{definition of } (\odot)\ \} \\
 & ((ys_1, n_1) \odot (ys, n)) \triangleright z.
\end{array}
$$

Therefore, one possible candidate of $(\odot)$ is

$$(ys_1, n_1) \odot (ys_2, n_2) = (\langle\!\langle X_1 \to X_2 +\!\!+ (X_3+)^*\ ys_2,\ (X_3+)^*\ ys_2 \rangle\!\rangle, X_3 + n_2),$$

which satisfies $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$ for any choice of X_1, X_2, and X_3.

Now we turn to the base case to figure out the rest of the metavariables. Expanding the base case,

$$(ys_1, n_1) = (ys_1, n_1) \odot ([\,], 0) = (\langle\!\langle X_1 \to X_2,\ [\,] \rangle\!\rangle, X_3),$$

one bold choice is letting $X_1 = \mathit{true}$, $X_2 = ys_1$, and $X_3 = n_1$. This completes our search for $(\odot)$:

$$(ys_1, n_1) \odot (ys_2, n_2) = (ys_1 +\!\!+ (n_1+)^*\ ys_2,\ n_1 + n_2).$$

6 Conclusions

Previous discussions on constructing list homomorphisms using the third list homomorphism theorem often overlook the fact that, for even slightly non-trivial problems, it is not an easy task to construct one of $(\triangleleft)$ and $(\triangleright)$, given another, in a constructive, formal manner, and thus an associativity proof is often needed. The $(\odot)$ operator, on the other hand, is a generalisation of $(\triangleleft)$ and $(\triangleright)$, whose proof of correctness also generalises from the proof of associativity. It is thus a waste throwing the proof away.

We have proposed and demonstrated a novel approach to constructing $(\odot)$. Starting with a trivial generalisation of either $(\triangleleft)$ or $(\triangleright)$, we exploit the constraint enforced by the proof of associativity to refine $(\odot)$. Once we have constructed $(\odot)$, we have its correctness proof too. It also explains the phenomena that in practice, $(\odot)$ often consists of fragments of code from $(\triangleleft)$ and $(\triangleright)$ — it can be constructed by generalising from one of them before being refined by another.

Acknowledgements. The authors would like to thank Zhenjiang Hu and Akimasa Morihata for discussion on the subject matter and, in particular, for suggesting the reference to work by Geser and Gorlatch. We would also like to thank the anonymous referees for their useful comments.

References

1. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. NATO ASI Series F, vol. 36, pp. 3–42. Springer, Heidelberg (1987)
2. Bird, R.S.: Introduction to Functional Programming using Haskell. Prentice-Hall (1998)
3. Bird, R.S., de Moor, O.: Algebra of Programming. International Series in Computer Science. Prentice-Hall (1997)
4. Cole, M.: Parallel programming, list homomorphisms and the maximum segment sum problems. Tech. Rep. CSR-25-93, Department of Computer Science, University of Edinburgh (1993)
5. Dijkstra, E.W.: The next fifty years. Tech. Rep. EWD1243, Eindhoven University of Technology (2004)
6. Geser, A., Gorlatch, S.: Parallelizing functional programs by generalization. Journal of Functional Programming 9(6), 649–673 (1999)
7. Gibbons, J.: The third homomorphism theorem. Journal of Functional Programming 6(4), 657–665 (1996)
8. Hu, Z., Iwasaki, H., Takeichi, M.: Construction of List Homomorphisms Via Tupling and Fusion. In: Penczek, W., Szałas, A. (eds.) MFCS 1996. LNCS, vol. 1113, pp. 407–418. Springer, Heidelberg (1996)
9. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In: Shao, Z., Pierce, B.C. (eds.) Symposium on Principles of Programming Languages, pp. 177–185. ACM Press, Savannah (2009)
10. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: Ferrante, J., McKinley, K.S. (eds.) Programming Language Design and Implementation, pp. 146–155. ACM Press (2007)

Extending Hindley-Milner Type Inference with Coercive Structural Subtyping

Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany

Abstract. We investigate how to add coercive structural subtyping to a type system for simply-typed lambda calculus with Hindley-Milner polymorphism. Coercions allow to convert between different types, and their automatic insertion can greatly increase readability of terms. We present a type inference algorithm that, given a term without type information, computes a type assignment and determines at which positions in the term coercions have to be inserted to make it type-correct according to the standard Hindley-Milner system (without any subtypes). The algorithm is sound and, if the subtype relation on base types is a disjoint union of lattices, also complete. The algorithm has been implemented in the proof assistant Isabelle.

1 Introduction

The main idea of subtype polymorphism, or simply subtyping, is to allow the programmer to omit type conversions, also called *coercions*. Inheritance in object-oriented programming languages can be viewed as a form of subtyping.

Although the ability to omit coercions is important to avoid unnecessary clutter in programs, subtyping is not a common feature in functional programming languages, such as ML or Haskell. The main reason for this is the increase in complexity of type inference systems with subtyping compared to Milner's well-known algorithm W [7]. In contrast, the theorem prover Coq supports coercive subtyping, albeit in an incomplete manner. Our contributions to this extensively studied area are:

- a comparatively simple type and coercion inference algorithm with
- soundness and completeness results improving on related work (see the beginning of §3 and the end of §4), and
- a practical implementation in the Isabelle theorem prover. This extension is very effective, for example, in the area of numeric types (*nat*, *int*, *real* etc), which require coercions that used to clutter up Isabelle text.

Our work does not change the standard Hindley-Milner type system (and hence leaves the Isabelle kernel unchanged!) but infers where coercions need to be inserted to make some term type correct.

The rest of this paper is structured as follows. In §2 we introduce terms, types, coercions and subtyping. §3 presents our type inference algorithm for

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 89–104, 2011.

simply-typed lambda calculus with coercions and Hindley-Milner polymorphism. In §4, we formulate the correctness and completeness statements and discuss restrictions on the subtype relation that are necessary to prove them. An outline of related research is given in §5.

2 Notation and Terminology

2.1 Terms and Types

The types and terms of simply-typed lambda calculus are given by the following grammars:

$$\tau = \alpha \mid T \mid C\ \tau\ \ldots\ \tau$$

$$t = x \mid c_{[\overline{\alpha}\mapsto\overline{\tau}]} \mid (\lambda x : \tau.\ t) \mid t\ t$$

A type can be a *type variable* (denoted by α, β, ...), a *base type* (denoted by S, T, U, ...), or a *compound type*, which is a type constructor (denoted by C, D, ...) applied to a list of type arguments. The number of arguments of a type constructor C, which must be at least one, is called the *arity* of C. The function type is a special case of a binary type constructor. We use the common infix notation $\tau \to \sigma$ in this case. Terms can be variables (denoted by x, y, ...), abstractions, or applications. In addition, a term can contain *constants* (denoted by c, d, ...) of polymorphic type. All terms are defined over a *signature* Σ that maps each constant to a *schematic type*, i.e. a type containing variables. In every occurrence of a constant c, the variables in its schematic type can be instantiated in a different way, for which we use the notation $c_{[\overline{\alpha}\mapsto\overline{\tau}]}$, where $\overline{\alpha}$ denotes the vector of free variables in the type of c (ordered in a canonical way), and $\overline{\tau}$ denotes the vector of types that the free variables are instantiated with. The type checking rules for terms are shown in Figure 1.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Ty-Var} \qquad \frac{\Sigma(c) = \sigma}{\Gamma \vdash c_{[\overline{\alpha}\mapsto\overline{\tau}]} : \sigma[\overline{\alpha} \mapsto \overline{\tau}]}\ \textsc{Ty-Const}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x : \tau.\ t : \tau \to \sigma}\ \textsc{Ty-Abs} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ t_2 : \sigma}\ \textsc{Ty-App}$$

Fig. 1. Type checking rules

2.2 Subtyping and Coercions

We write $\tau <: \sigma$ to denote that τ is a *subtype* of σ. The subtyping relation that we consider in this paper is *structural*: if $\tau <: \sigma$, then τ and σ can only differ in their base types. For example, we may have $C\ T <: C\ U$, but not $C\ T <: S$. Type checking rules for systems with subtypes are often presented using a so-called *subsumption* rule

$$\frac{\Gamma \vdash t : \tau \qquad \tau <: \sigma}{\Gamma \vdash t : \sigma}$$

allowing a term t of type τ to be used in a context where a term of the supertype σ would be expected. The problem of deciding whether a term is typable using the subsumption rule is equivalent to the problem of deciding whether this term can be made typable without the subsumption rule by inserting coercion functions in appropriate places in the term. Rather than extending our type system with a subsumption rule, we therefore introduce a new judgement $\Gamma \vdash t \rightsquigarrow u : \tau$ that, given a context Γ and a term t, returns a new term u augmented with coercions, together with a type τ, such that $\Gamma \vdash u : \tau$ holds. We write $\tau <:_c \sigma$ to mean that c is a coercion of type $\tau \to \sigma$. Coercions can be built up from a set of coercions $\mathcal{C}$ between base types, and from a set of *map functions* $\mathcal{M}$ for building coercions between constructed types from coercions between their argument types as shown in Figure 2. The sets $\mathcal{C}$ and $\mathcal{M}$ are parameters of our setup. We restrict $\mathcal{M}$ to contain at most one map function for a type constructor.

Definition 1 (Map function). *Let C be an n-ary type constructor. A function f of type*

$$\tau_1 \to \cdots \to \tau_n \to C\ \alpha_1\ \ldots\ \alpha_n \to C\ \beta_1\ \ldots\ \beta_n$$

where $\tau_i \in \{\alpha_i \to \beta_i, \beta_i \to \alpha_i\}$, is called a map function *for C. If $\tau_i = \alpha_i \to \beta_i$, then C is called* covariant *in the i-th argument wrt. f, otherwise* contravariant.

$$\frac{}{\tau <:_{id} \tau}\ \textsc{Gen-Refl} \qquad \frac{\Sigma(c) = T \to U \qquad c \in \mathcal{C}}{T <:_c U}\ \textsc{Gen-Base}$$

$$\frac{T <:_{c_1} U \qquad U <:_{c_2} S}{T <:_{\lambda x:T.c_2\ (c_1\ x)} S}\ \textsc{Gen-Trans}$$

$$\frac{\begin{array}{c} map_C : (\delta_1 \to \rho_1) \to \cdots \to (\delta_n \to \rho_n) \to C\ \alpha_1 \ldots \alpha_n \to C\ \beta_1 \ldots \beta_n \in \mathcal{M} \\ \theta = \{\overline{\alpha} \mapsto \overline{\tau}, \overline{\beta} \mapsto \overline{\sigma}\} \qquad \forall 1 \le i \le n.\ \theta(\delta_i) <:_{c_i} \theta(\rho_i) \end{array}}{C\ \tau_1 \ldots \tau_n <:_{\theta(map_C\ c_1\ \ldots\ c_n)} C\ \sigma_1 \ldots \sigma_n}\ \textsc{Gen-Cons}$$

Fig. 2. Coercion generation

For the implementation of type checking and inference algorithms, the subsumption rule is problematic, because it is not syntax directed. However, it can be shown that any derivation of $\Gamma \vdash t : \sigma$ using the subsumption rule can be transformed into a derivation of $\Gamma \vdash t : \tau$ with $\tau <: \sigma$, in which the subsumption rule is only applied to function arguments [12, §16.2]. Consequently, the coercion insertion judgement shown in Figure 3 only inserts coercions in argument positions of functions by means of the Coerce-App rule.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau}\ \textsc{Coerce-Var} \qquad \frac{\Sigma(c) = \sigma}{\Gamma \vdash c_{[\overline{\alpha}\mapsto\overline{\tau}]} \rightsquigarrow c_{[\overline{\alpha}\mapsto\overline{\tau}]} : \sigma[\overline{\alpha} \mapsto \overline{\tau}]}\ \textsc{Coerce-Const}$$

$$\frac{\Gamma, x : \tau \vdash t \rightsquigarrow u : \sigma}{\Gamma \vdash \lambda x : \tau.\ t \rightsquigarrow \lambda x : \tau.\ u : \tau \to \sigma}\ \textsc{Coerce-Abs}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow u_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash t_2 \rightsquigarrow u_2 : \tau_2 \qquad \tau_2 <:_c \tau_{11}}{\Gamma \vdash t_1\ t_2 \rightsquigarrow u_1\ (c\ u_2) : \tau_{12}}\ \textsc{Coerce-App}$$

Fig. 3. Coercion insertion

2.3 Type Substitutions and Unification

A central component of type inference systems is a *unification* algorithm for types. Implementing such an algorithm for the type expressions introduced in §2.1 is straightforward, since this is just an instance of first-order unification. We write *mgu* for the function computing the most general unifier. It produces a *type substitution*, denoted by θ, which is a function mapping type variables to types such that $\theta\alpha \neq \alpha$ for only finitely many α. We will sometimes use the notation $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ to denote such substitutions. Type substitutions are extended to types, terms, and any other data structures containing type variables in the usual way. The function *mgu* is overloaded: it can be applied to pairs of terms, where $\theta\tau = \theta\sigma$ if $\theta = mgu(\tau, \sigma)$, to (finite) sets of equality constraints, where $\theta\tau_i = \theta\sigma_i$ if $\theta = mgu\{\tau_1 \doteq \sigma_1, \ldots, \tau_n \doteq \sigma_n\}$, as well as to (finite) sets of types, where $\theta\tau_1 = \cdots = \theta\tau_n$ if $\theta = mgu\{\tau_1, \ldots, \tau_n\}$.

3 Type Inference with Coercions

In a system without coercions, *type inference* means to find a type substitution θ and a type τ for a given term t and context Γ such that t becomes typable, i.e. $\theta\Gamma \vdash \theta t : \tau$. In a system with coercions, type inference also has to insert coercions into the term t in appropriate places, yielding a term u for which $\theta\Gamma \vdash \theta t \rightsquigarrow u : \tau$ and $\theta\Gamma \vdash u : \tau$ holds. A naive way of doing type inference in this setting would be to compute the substitution θ and insert the coercions on-the-fly, as suggested by Luo [6]. The idea behind Luo's type inference algorithm is to try to do standard Hindley-Milner type inference first, and locally repair typing problems by inserting coercions only if the standard algorithm fails. However, this approach has a serious drawback: the success or failure of the algorithm depends on the order in which the types of subterms are inferred. To see why this is the case, consider the following example.

Example 1. Let $\Sigma = \{leq : \alpha \to \alpha \to \mathbb{B}, n : \mathbb{N}, i : \mathbb{Z}\}$ be the signature containing a polymorphic predicate *leq* (e.g. less-or-equal), as well as a natural number constant n

and an integer constant i. Moreover, assume that the set of coercions $\mathcal{C} = \{int : \mathbb{N} \to \mathbb{Z}\}$ contains a coercion from natural numbers to integers, but not from integers to natural numbers, since this would cause a loss of information. Then, it is easy to see that the terms $leq_{[\alpha \mapsto \beta]}\ i\ n$ and $leq_{[\alpha \mapsto \beta]}\ n\ i$ can both be made type correct by applying the type substitution $\{\beta \mapsto \mathbb{Z}\}$ and inserting coercions, but the naive algorithm can only infer the type of the first term. Since the term is an application, the algorithm would first infer (using standard Hindley-Milner type inference) that the function denoted by the subterm $leq_{[\alpha \mapsto \beta]}\ i$ has type $\mathbb{Z} \to \mathbb{B}$ with the type substitution $\{\beta \mapsto \mathbb{Z}\}$. Similarly, for the subterm n the type $\mathbb{N}$ is inferred. Since the argument type $\mathbb{Z}$ of the function does not match the type $\mathbb{N}$ of its argument, the algorithm inserts the coercion int to repair the typing problem, yielding the term $leq_{[\alpha \mapsto \mathbb{Z}]}\ i\ (int\ n)$ with type $\mathbb{B}$. In contrast, when inferring the type of the term $leq_{[\alpha \mapsto \beta]}\ n\ i$, the algorithm would first infer that the subterm $leq_{[\alpha \mapsto \beta]}\ n$ has type $\mathbb{N} \to \mathbb{B}$, using the type substitution $\{\beta \mapsto \mathbb{N}\}$. The subterm i is easily seen to have type $\mathbb{Z}$, which does not match the argument type $\mathbb{N}$ of the function. However, in this case, the type mismatch cannot be repaired, since there is no coercion from $\mathbb{Z}$ to $\mathbb{N}$, and so the algorithm fails.

The strategy for coercion insertion used in the Coq proof assistant (originally due to Saïbi [15], who provides no soundness or completeness results) suffers from similar problems, which the reference manual describes as the "normal" behaviour of coercions [3, §17.12]. Our goal is to provide a complete algorithm that does not fail in cases such as the above.

3.1 Coercive Subtyping Using Subtype Constraints

The algorithm presented here generates subtype constraints first, and postpones their solution as well as the insertion of coercions to a later stage of the algorithm. The set of all constraints provides us with a global view on the term that we are processing, and therefore avoids the problems of a local algorithm.

The algorithm can be divided into four major phases. First, we generate subtype constraints by recursively traversing the term. Then, we simplify these constraints, which can be inequalities between arbitrary types, until the constraint set contains only inequalities between base types and variables. The next step is to organize these *atomic* constraints in a graph and solve them, which means to find a type substitution. Applying this substitution to the whole constraint set results in inequalities that are consistent with the given partial order on base types. Finally, the coercions are inserted by traversing the term for the second time. A visualization of the main steps of the algorithm in form of a control flow is shown in Figure 4.

3.2 Constraint Generation

The algorithm for constraint generation is described by a judgement $\Gamma \vdash t : \tau \triangleright S$ defined by the rules shown in Figure 5. Given a term t and a context Γ, the algorithm returns a type τ, as well as a set of *equality* and *subtype constraints* S denoted by infix "$\doteq$" and "$<:$", respectively. The equality constraints are solved using unification, whereas the subtype constraints are simplified to atomic

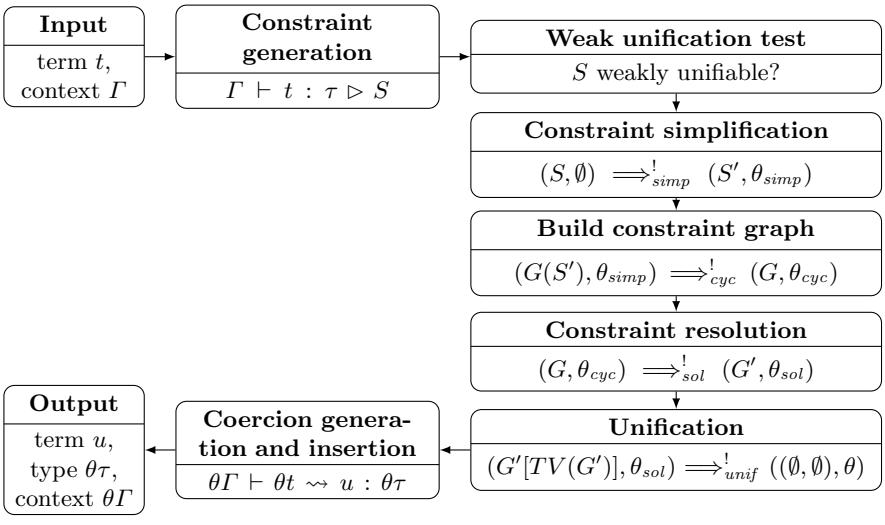


Fig. 4. Top-level control flow of the subtyping algorithm

constraints and then solved using the graph-based algorithm mentioned above. The only place where new constraints are generated is the rule SubCT-App for function applications $t_1\ t_2$. It generates an equality constraint ensuring that the type of t_1 is actually a function type, as well as a subtype constraint ensuring that the type of t_2 is a subtype of the argument type of t_1.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rhd \emptyset}\ \textsc{SubCT-Var} \qquad \frac{\Sigma(c) = \sigma}{\Gamma \vdash c_{[\overline{\alpha} \mapsto \overline{\tau}]} : \sigma[\overline{\alpha} \mapsto \overline{\tau}] \rhd \emptyset}\ \textsc{SubCT-Const}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma \rhd S}{\Gamma \vdash \lambda x : \tau.\ t : \tau \to \sigma \rhd S}\ \textsc{SubCT-Abs}$$

$$\frac{\Gamma \vdash t_1 : \tau \rhd S_1 \qquad \Gamma \vdash t_2 : \sigma \rhd S_2 \qquad \alpha, \beta \text{ fresh}}{\Gamma \vdash t_1\ t_2 : \beta \rhd S_1 \cup S_2 \cup \{\tau \doteq \alpha \to \beta, \sigma <: \alpha\}}\ \textsc{SubCT-App}$$

Fig. 5. Constraint generation rules

Note that as a first step not shown here, the type-free term input by the user is augmented with type variables: $\lambda x.\ t$ becomes $\lambda x : \beta.\ t$ and c becomes $c_{[\overline{\alpha} \mapsto \overline{\beta}]}$, where all the βs must be distinct and new.

3.3 Constraint Simplification

The constraints generated in the previous step are now simplified by repeatedly applying the transformation rules shown in Figure 6. The states that the

transformation operates on are pairs whose first component contains the current set of constraints, while the second component is used to accumulate the substitutions computed during the transformation. As a starting state of the transformation, we use the pair $(S, \emptyset)$. The rule DECOMPOSE splits up inequations between complex types into simpler inequations or equations according to the *variance* of the outermost type constructor. For this purpose, we introduce a *variance operator*, which is defined as follows.

Definition 2 (Variance operator). *Let map_C be a map function for the type constructor C of arity n in the set $\mathcal{M}$. We use the abbreviation*

$$var^i_C(\tau, \sigma) = \begin{cases} \tau <: \sigma & \text{if } C \text{ is covariant in the } i\text{-th argument wrt. } map_C \\ \sigma <: \tau & \text{if } C \text{ is contravariant in the } i\text{-th argument wrt. } map_C \end{cases}$$

for $1 \leq i \leq n$. If there is no such map_C, then we define for $1 \leq i \leq n$:

$$var^i_C(\tau, \sigma) = \tau \doteq \sigma.$$

Thus, if no map function is associated with a particular type constructor, it is considered to be *invariant*, causing the algorithm to generate equations instead of inequations. Equations are dealt with by rule UNIFY using ordinary unification. Since our subtyping relation is structural, an inequation having a type variable on one side, and a complex type on the other side can only be solved by instantiating the type variable with a type whose outermost type constructor equals that of the complex type on the other side. This is expressed by the two symmetric rules EXPAND-L and EXPAND-R. Finally, inequations with an atomic type on both sides are eliminated by rule ELIMINATE, provided they conform to the subtyping relation.

We apply these rules repeatedly to the constraint set until none of the rules is applicable. Therefore, we use the notation $\Longrightarrow^!_{simp}$.

Definition 3. *(Normal form) For a relation $\Longrightarrow$ we write*

$$X \Longrightarrow^! X'$$

if $X \Longrightarrow^* X'$ and X' is in normal form *wrt. $\Longrightarrow$.*

Definition 4 (Atomic constraint). *We call a subtype constraint* atomic *if it corresponds to one of the following constraints (α, β are type variables, T is a base type):*

$$\alpha <: \beta \qquad \alpha <: T \qquad T <: \alpha$$

If none of the rules is applicable, the algorithm terminates in a state (S', θ_{simp}), where S' either consists only of atomic constraints, or S' contains an inequation $C_1\ \overline{\tau} <: C_2\ \overline{\sigma}$ with $C_1 \neq C_2$ or an inequation $T <: U$ for base types T and U such that T is not a subtype of U or an equation $\tau \doteq \sigma$ such that τ and σ are not unifiable. In the latter three cases, the type inference algorithm fails.

DECOMPOSE
$(\{C\ \tau_1\ \ldots\ \tau_n <: C\ \sigma_1\ \ldots\ \sigma_n\} \uplus S, \theta) \Longrightarrow_{simp} \left(\left\{var_C^i\,(\tau_i, \sigma_i) \mid i = 1 \ldots n\right\} \cup S, \theta\right)$

UNIFY
$(\{\tau \doteq \sigma\} \uplus S, \theta) \Longrightarrow_{simp} (\theta' S, \theta' \circ \theta)$
where $\theta' = mgu(\tau, \sigma)$

EXPAND-L
$(\{\alpha <: C\ \tau_1\ \ldots\ \tau_n\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'\,(\{\alpha <: C\ \tau_1\ \ldots\ \tau_n\} \cup S), \theta' \circ \theta)$
where $\theta' = \{\alpha \mapsto C\ \alpha_1\ \ldots\ \alpha_n\}$
and $\alpha_1\ \ldots\ \alpha_n$ are fresh variables

EXPAND-R
$(\{C\ \tau_1\ \ldots\ \tau_n <: \alpha\} \uplus S, \theta) \Longrightarrow_{simp} (\theta'\,(\{C\ \tau_1\ \ldots\ \tau_n <: \alpha\} \cup S), \theta' \circ \theta)$
where $\theta' = \{\alpha \mapsto C\ \alpha_1\ \ldots\ \alpha_n\}$
and $\alpha_1\ \ldots\ \alpha_n$ are fresh variables

ELIMINATE
$(\{U <: T\} \uplus S, \theta) \Longrightarrow_{simp} (S, \theta)$
where U, T are base types
and $U <: T$

Fig. 6. Rule-based constraint simplification $\Longrightarrow_{simp}$

An interesting question is whether such a state or a failure is always reached after a finite number of iterations. It is obvious that the simplification of the constraint $\alpha <: C\ \alpha$ will never terminate. Bourdoncle and Merz [2] have pointed out that checking whether the initial constraint set has a *weak unifier* is sufficient to avoid nontermination. Weak unification differs from standard unification in that it identifies base types, which is necessary since two types τ and σ with $\tau <: \sigma$ need to be equal up to their base types.

Definition 5 (Weak unification). *A set of constraints S is called* weakly unifiable *if there exists a substitution θ such that $\lceil\theta\tau\rceil = \lceil\theta\sigma\rceil$ for all $\tau <: \sigma \in S$, and $\theta\tau = \theta\sigma$ for all $\tau \doteq \sigma \in S$, where*

$$\begin{aligned} \lceil\alpha\rceil &= \alpha \\ \lceil T\rceil &= T_0 \\ \lceil C\ \tau_1\ \ldots\ \tau_n\rceil &= C\ \lceil\tau_1\rceil\ \ldots\ \lceil\tau_n\rceil \end{aligned}$$

and T_0 is a fixed base type not used elsewhere.

Weak unification is merely used as a termination-test in our algorithm before constraint simplification (see Figure 4).

3.4 Solving Subtype Constraints on a Graph

An efficient and logically clean way to reason about atomic subtype constraints is to represent the types as nodes of a directed graph with arcs given by the

constraints themselves. Concretely, this means that a subtype constraint $\sigma <: \tau$ is represented by the arc (σ, τ). This allows us to speak of predecessors and successors of a type.

Definition 6 (Constraint graph). *For a constraint set S, we denote by*

$$G(S) = (\bigcup\{\{\tau, \sigma\} \mid \tau <: \sigma \in S\}, \{(\tau, \sigma) \mid \tau <: \sigma \in S\})$$

the constraint graph *corresponding to S.*

Given a graph $G = (V, E)$, the subgraph induced by a vertex set $X \subseteq V$ is denoted by $G[X] = (X, (X \times X) \cap E)$. The set of type variables contained in the vertex set of G is denoted by $TV(G)$.

In what follows, we write $\sigma \preceq: \tau$ for the subtyping relation on base types induced by the set of coercions $\mathcal{C}$, which is defined by

$$\preceq: \;=\; \{(T, U) \mid c : T \to U \in \mathcal{C}\}^*$$

Graph Construction. Building such a constraint graph is straightforward. We only need to watch out for cycles. Since the subtype relation is a partial order and therefore antisymmetric, at most one base type should occur in a cycle. In other words, if the elements of the cycle are not unifiable, the inference will fail. Unifiable cycles should be eliminated with the iterated application of the rule Cycle-Elim shown in Figure 7.

Cycle-Elim

$$((V, E), \theta) \Longrightarrow_{cyc} ((V \setminus K \cup \{\tau_K\}, E' \cup P \times \{\tau_K\} \cup \{\tau_K\} \times S), \theta_K \circ \theta)$$

where K is a cycle in (V, E)
and $\theta_K = mgu(K)$
and $\{\tau_K\} = \theta_K K$
and $E' = \{(\tau, \sigma) \in E \mid \tau \notin K, \sigma \notin K\}$
and $P = \{\tau \mid \exists \sigma \in K.\ (\tau, \sigma) \in E\} \setminus K$
and $S = \{\sigma \mid \exists \tau \in K.\ (\tau, \sigma) \in E\} \setminus K$

Fig. 7. Rule-based cycle elimination $\Longrightarrow_{cyc}$

Figure 8 visualizes an example of cycle elimination. We call the substitution obtained from cycle elimination θ_{cyc}.

Constraint Resolution. Now we must find an assignment for all variables that appear in the graph $G = (V, E)$. We use an algorithm that is based on the approach presented in [20]. First, we define some basic lattice-theoretic notions.

Definition 7. *Let S, T, T' denote base types and X a set of base types. With respect to the given subtype relation $\preceq:$ we define:*

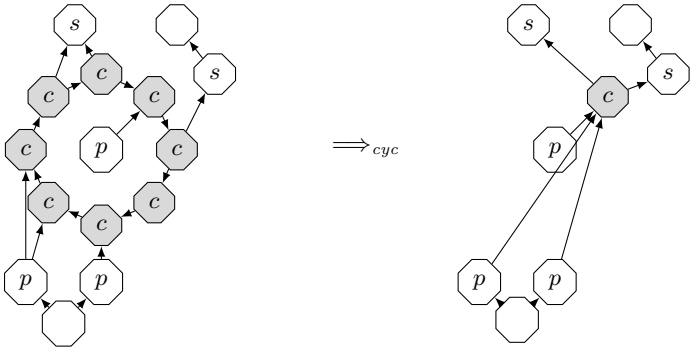


Fig. 8. Collapse of a cycle in a graph

- $\overline{T} = \{T' \mid T \preceq: T'\}$, *the set of* supertypes
- $\underline{T} = \{T' \mid T' \preceq: T\}$, *the set of* subtypes
- $T \sqcup S \in \overline{T} \cap \overline{S}$ *and* $\forall U \in \overline{T} \cap \overline{S}.\ T \sqcup S \preceq: U$, *the* supremum *of* S *and* T
- $T \sqcap S \in \underline{T} \cap \underline{S}$ *and* $\forall L \in \underline{T} \cap \underline{S}.\ L \preceq: T \sqcap S$, *the* infimum *of* S *and* T
- $\bigsqcup X \in \bigcap_{T \in X} \overline{T}$ *and* $\forall U \in \bigcap_{T \in X} \overline{T}.\ \bigsqcup X \preceq: U$, *the* supremum *of* X
- $\bigsqcap X \in \bigcap_{T \in X} \underline{T}$ *and* $\forall L \in \bigcap_{T \in X} \underline{T}.\ L \preceq: \bigsqcap X$, *the* infimum *of* X.

Note that, depending on $\preceq:$, *suprema or infima may not exist.*

Given a type variable α *in the constraint graph* $G = (V, E)$, *we define:*

- $P^G_\alpha = \{T \mid (T, \alpha) \in E^+\}$, *the set of all base type* predecessors *of* α
- $S^G_\alpha = \{T \mid (\alpha, T) \in E^+\}$, *the set of all base type* successors *of* α.

E^+ *is the transitive closure of the edges of* G.

The algorithm assigns base types to type variables that have base type successors or predecessors until no such variables are left using the rules shown in Figure 9. The resulting substitution is referred to as θ_{sol}.

The original algorithm described by Wand and O'Keefe [20] is designed to be complete for subtype relations that form a tree. It only uses the rules Assign-Inf and Fail-Inf without the check if S^G_α is empty. It assigns each type variable α the infimum $\bigsqcap S^G_\alpha$ of its upper bounds, and then checks whether the assigned type is greater than all lower bounds P^G_α. If $\bigsqcap S^G_\alpha$ does not exist, their algorithm fails. If S^G_α is empty, its infimum only exists if there is a greatest type, which exists in a tree but not in a forest. In order to avoid this failure in the absence of a greatest type, our algorithm does not compute the infimum/supremum of the empty set, and is symmetric in successors/predecessors.

After constraint resolution, unassigned variables can only occur in the resulting graph in weakly connected components that do not contain any base types. As we do not want to annotate the term with unresolved subtype constraints, all variables in a single weakly connected component should be unified. This is done by the rule Unify-WCC shown in Figure 10 and produces the final substitution θ.

$$\textsc{Assign-Sup}\quad (G,\theta) \Longrightarrow_{sol} \big(\{\alpha \mapsto \bigsqcup P_\alpha^G\}G, \{\alpha \mapsto \bigsqcup P_\alpha^G\} \circ \theta\big) \text{ if } \alpha \in TV(G) \wedge P_\alpha^G \neq \emptyset \wedge \exists \bigsqcup P_\alpha^G \wedge \forall T \in S_\alpha^G.\ \bigsqcup P_\alpha^G \preceq: T$$

$$\textsc{Fail-Sup}\quad (G,\theta) \Longrightarrow_{sol} \text{FAIL} \text{ if } \alpha \in TV(G) \wedge P_\alpha^G \neq \emptyset \wedge (\nexists \bigsqcup P_\alpha^G \vee \exists T \in S_\alpha^G.\ \bigsqcup P_\alpha^G \not\preceq: T)$$

$$\textsc{Assign-Inf}\quad (G,\theta) \Longrightarrow_{sol} \big(\{\alpha \mapsto \bigsqcap S_\alpha^G\}G, \{\alpha \mapsto \bigsqcap S_\alpha^G\} \circ \theta\big) \text{ if } \alpha \in TV(G) \wedge S_\alpha^G \neq \emptyset \wedge \exists \bigsqcap S_\alpha^G \wedge \forall T \in P_\alpha^G.\ T \preceq: \bigsqcap S_\alpha^G$$

$$\textsc{Fail-Inf}\quad (G,\theta) \Longrightarrow_{sol} \text{FAIL} \text{ if } \alpha \in TV(G) \wedge S_\alpha^G \neq \emptyset \wedge (\nexists \bigsqcap S_\alpha^G \vee \exists T \in P_\alpha^G.\ T \not\preceq: \bigsqcap S_\alpha^G)$$

Fig. 9. Rule-based constraint resolution $\Longrightarrow_{sol}$

$$\textsc{Unify-WCC}\quad (G,\theta) \Longrightarrow_{unif} (G[V \setminus W], mgu(W) \circ \theta) \text{ where } W \text{ is a weakly connected component of } G = (V,E)$$

Fig. 10. Rule-based WCC-unification $\Longrightarrow_{unif}$

Example 2. Going back to Example 1, we apply our algorithm to the term $leq_{[\alpha \mapsto \alpha_3]}\ n\ i$. According to the inference rules from Figure 5, we obtain $\Gamma \vdash leq_{[\alpha \mapsto \alpha_3]}\ n\ i : \beta_1 \rhd \{\alpha_3 \to \alpha_3 \to \mathbb{B} \doteq \alpha_2 \to \beta_2, \beta_2 \doteq \alpha_1 \to \beta_1, \mathbb{N} <: \alpha_2, \mathbb{Z} <: \alpha_1\}$. Simplifying the generated constraints yields the substitution $\theta_{simp} = \{\alpha_1 \mapsto \alpha_3, \alpha_2 \mapsto \alpha_3, \beta_1 \mapsto \mathbb{B}, \beta_2 \mapsto \alpha_3 \to \mathbb{B}\}$ and the atomic constraint set $\{\mathbb{N} <: \alpha_3, \mathbb{Z} <: \alpha_3\}$. This yields the constraint graph shown in Figure 11. The constraint resolution algorithm assigns α_3 the least upper bound of $\{\mathbb{N}, \mathbb{Z}\}$, which is $\mathbb{Z}$. The resulting substitution is $\theta_{sol} = \{\alpha_1 \mapsto \mathbb{Z}, \alpha_2 \mapsto \mathbb{Z}, \alpha_3 \mapsto \mathbb{Z}, \beta_1 \mapsto \mathbb{B}, \beta_2 \mapsto \mathbb{Z} \to \mathbb{B}\}$. Since there are no unassigned variables in the remaining constraint graph, Unify-WCC is inapplicable and θ, the final result, is θ_{sol}.

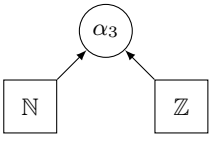


Fig. 11. Constraint graph of $leq_{[\alpha \mapsto \alpha_3]}\ n\ i$

In §4 we will see that the constraint resolution algorithm defined in this subsection is not complete in general but is complete if the partial order on base types is a disjoint union of lattices.

3.5 Coercion Insertion

Finally, we have a solving substitution θ. Applying this substitution to the initial term will produce a term that can always be coerced to a type correct term by means of the coercion insertion judgement shown in Figure 3. We inspect this correctness statement and the termination of our algorithm in §4.

4 Total Correctness and Completeness

To prove total correctness, we need to show that for any input t and Γ, the algorithm either returns a substitution θ, a well-typed term u together with its type $\theta\tau$ or indicates a failure. Failures may occur at any computation of a most general unifier, during the weak unification test, or explicitly at the reduction steps Fail-Sup and Fail-Inf in the constraint resolution phase. Below we discuss correctness and termination. Due to space limitations, all proofs and supporting lemmas had to be omitted and can be found in the full version of this paper [19]. Since the reduction rules in each phase are applied nondeterministically, the algorithm may output different substitutions for the same input term t and context Γ. By $\mathit{AlgSol}(\Gamma, t)$ we denote the set of all such substitutions.

Theorem 1 (Correctness). *For a given term t in the context Γ, assume $\theta \in \mathit{AlgSol}(\Gamma, t)$. Then there exist a term u and a type τ, such that $\theta\Gamma \vdash \theta t \rightsquigarrow u : \tau$ and $\theta\Gamma \vdash u : \tau$.*

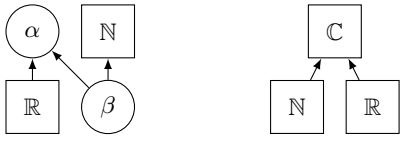
Thus, we know that if the algorithm terminates successfully, it returns a well-typed term. Moreover, it terminates for any input:

Theorem 2 (Termination). *The algorithm terminates for any input t and Γ.*

4.1 An Example for Incompleteness

So far, we have only made statements about termination and correctness of our algorithm. It is equally important that the algorithm does not fail for a term that can be coerced to a well-typed term. An algorithm with this property is called complete. As mentioned earlier, our algorithm is not complete for arbitrary posets of base types.

Example 3. Figure 12 shows a constraint graph and base type order where our algorithm may fail, although $\{\alpha \mapsto \mathbb{C},\ \beta \mapsto \mathbb{N}\}$ is a solving substitution. If during constraint resolution the type variable α is assigned first, it will receive value $\mathbb{R}$. Then, the assignment of β will fail, since the infimum $\mathbb{R} \sqcap \mathbb{N}$ does not exist in the given poset. The fact that our algorithm does find the solution if β is assigned before α is practically irrelevant because we cannot possibly exhaust all nondeterministic choices.



Constraint graph Partial order on base types

Fig. 12. Problematic example without type classes

We see that the problem is the non-existence of a supremum or infimum. The solution to this problem is to require a certain lattice structure for the partial order on base types. Alternatively we could try and generalize our algorithm, but this is unappealing for complexity theoretic reasons.

4.2 Complexity and Completeness

Tiuryn and Frey [18,4] showed that the general constraint satisfaction problem is PSPACE-complete. Tiuryn [18] also shows that satisfiability can be tested in polynomial time if the partial order on base types is a disjoint union of lattices. Unfortunately, Tiuryn only gives a decision procedure that does not compute a solution. Nevertheless, most if not all approaches in the literature adopt the restriction to (disjoint unions of) lattices, but propose algorithms that are exponential in the worst case. This paper is no exception. Just like Simonet [17] we argue that in practice the exponential nature of our algorithm does not show up. Our implementation in Isabelle confirms this.

All phases of our algorithm have polynomial complexity except for constraint simplification: a cascade of applications of Expand-L or Expand-R may produce an exponential number of new type variables. Restricting to disjoint union of lattices does not improve the complexity but guarantees completeness of our algorithm because it guarantees the existence of the necessary infima and suprema for constraint resolution.

Therefore, we assume in the following that the base type poset is a disjoint union of lattices.

To formulate the completeness theorem, we need some further notation.

Definition 8 (Equality modulo coercions). *Two substitutions θ and θ' are* equal modulo coercions *wrt. the type variable set X, if for all $x \in X$ there exists a coercion c such that either $\theta(x) <:_c \theta'(x)$ or $\theta'(x) <:_c \theta(x)$ holds. We write $\theta \approx_X \theta'$.*

Definition 9 (Subsumed). *The substitution θ' is* subsumed modulo coercions *wrt. to the type variable set X by the substitution θ, if there exists a substitution δ such that $\theta' \approx_X \delta \circ \theta$. We write $\theta \lesssim_X \theta'$.*

Let $TV(\tau)$ and $TV(t)$ be the sets of type variables that occur in τ and the type annotations of t. For a context $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, we denote by $TV(\Gamma)$ the set $\bigcup_{i=1}^{n} TV(\tau_i)$.

Theorem 3 (Completeness). *If* $\theta'\Gamma \vdash \theta' t \rightsquigarrow u : \tau'$*, then* $AlgSol(\Gamma, t) \neq \emptyset$ *and for all* $\theta \in AlgSol(\Gamma, t)$ *it holds that* $\theta \lesssim_{TV(\Gamma)\cup TV(t)} \theta'$*.*

It is instructive to consider a case where our algorithm is not able to reconstruct a particular substitution but only a subsumed one.

Example 4. Let $\Sigma = \{id : \alpha \to \alpha, n : \mathbb{N}, sin : \mathbb{R} \to \mathbb{R}\}$ be a signature and let $\mathcal{C} = \{int : \mathbb{N} \to \mathbb{Z}, real : \mathbb{Z} \to \mathbb{R}\}$ be a set of coercions. Now consider the term $sin\ (id_{[\alpha\mapsto\alpha_1]}\ n)$ in the empty context. The constraint resolution phase will be given the atomic constraints $\{\mathbb{N} <: \alpha_1, \alpha_1 <: \mathbb{R}\}$ and will assign α_1 the tightest bound either with respect to its predecessors or its successors: $AlgSol(\emptyset, sin\ (id_{[\alpha\mapsto\alpha_1]}\ n)) = \{\{\alpha_1 \mapsto \mathbb{N}\}, \{\alpha_1 \mapsto \mathbb{R}\}\}$.

The substitution $\{\alpha_1 \mapsto \mathbb{Z}\}$ is also solution of the typing problem, i.e. $\{\alpha_1 \mapsto \mathbb{Z}\}\emptyset \vdash \{\alpha_1 \mapsto \mathbb{Z}\}(sin\ (id_{[\alpha\mapsto\alpha_1]}\ n)) \rightsquigarrow sin\ (real\ (id_{[\alpha\mapsto\mathbb{Z}]}\ (int\ n))) : \mathbb{R}$. It is itself not a possible output of the algorithm, but it is subsumed modulo coercions by both of the substitutions that the algorithm can return.

The completeness theorem tells us that the algorithm never fails if there is a solution. The example shows us that the algorithm may fail to produce some particular solution. The completeness theorem also tells us that any solution is an instance of the computed solution, but only up to coercions. In practice this means that the user may have to provide some coercions (or type annotations) explicitly to obtain what she wants. This is not the fault of the algorithm but is unavoidable if the underlying type system does not provide native subtype constraints.

Compared with the work by Saïbi we have a completeness result. On the other hand he goes beyond coercions between atomic types, something we have implemented but not yet released. Luo also proves a completeness result, but his point of reference is a modified version of the Hindley-Milner system where coercions are inserted on the fly, which is weaker than our inference system. In most other papers the type system comes with subtype constraints built in (not an option for us) and unrestricted completeness results can be obtained.

5 Related Work

Type inference with automatic insertion of coercions in the context of functional programming languages was first studied by Mitchell [8,9]. First algorithms for type inference with subtypes were described by Fuh and Mishra [5] as well as Wand and O'Keefe [20]. The algorithm for constraint simplification presented in this paper resembles the MATCH algorithm by Fuh and Mishra. However, in order to avoid nontermination due to cyclic substitutions, they build up an extra data structure representing equivalence classes of atomic types, whereas we use a weak unification check suggested by Bourdoncle and Merz [2]. The seemingly simple problem of solving atomic subtype constraints has also been the subject of extensive studies. In their paper [5], Fuh and Mishra also describe a second algorithm CONSISTENT for solving this problem, but they do not mention any conditions for the subtype order on atomic types, so it is unclear whether their algorithm works in general. Pottier [13] describes a sound but incomplete simplification procedure for subtype constraints. Simonet [17] presents general

subtype constraint solvers and simplifiers for lattices designed for practical efficiency. Benke [1], as well as Pratt and Tiuryn [14] study the complexity of solving atomic constraints for a variety of different subtype orders. Extensions of Haskell with subtyping have been studied by Shields and Peyton Jones [16], as well as Nordlander [11].

5.1 Conclusion

Let us close with a few remarks on the realization of our algorithm in Isabelle. The abstract algorithm returns a set of results because coercion inference is ambiguous. For example, the term $sin(n+n)$, where $+ : \alpha \to \alpha \to \alpha$, $sin : \mathbb{R} \to \mathbb{R}$ and $n : \mathbb{N}$ has two type-correct completions with the coercion $real : \mathbb{N} \to \mathbb{R}$: $sin(real(n + n))$ and $sin(real\ n + real\ n)$. Our deterministic implementation happens to produce the first one. If the user wanted the second term, he would have to insert at least one *real* coercion. Because Isabelle is a theorem prover and because we did not modify its kernel, we do not have to worry whether the two terms are equivalent (this is known as coherence): in the worst case the system picks the wrong term and the proof one is currently engaged in fails or proves a different theorem, but it will still be a theorem.

To assess the effectiveness of our algorithm, we picked a representative Isabelle theory from real analysis (written at the time when all coercions had to be present) and removed as many coercions from it as our algorithm would allow — remember that some coercions may be needed to resolve ambiguity. Of 1061 coercions, only 221 remained. In contrast, the on-the-fly algorithm by Saïbi and Luo (see the beginning of §3) still needs 666 coercions. The subtype lattice in this theory is a linear order of the 3 types *nat*, *int*, *real*.

Isabelle supports an extension of Hindley-Milner polymorphism with type classes [10]. In the full version of this paper [19], we cover type classes, too, and show how to extend our algorithms soundly; completeness seems difficult to achieve in this context.

We have not mentioned *let* so far because it does not mesh well with coercive subtyping. Consider the term $t = let\ f = s\ in\ u$ where $u = (Suc(f(0)), f(0.0))$, $0 : \mathbb{N}$, $Suc : \mathbb{N} \to \mathbb{N}$, $0.0 : \mathbb{R}$, and s is a term that has type $\alpha \to \alpha$ under the constraints $\{\alpha \preceq: \mathbb{R},\ \alpha \preceq: \beta,\ \mathbb{N} \preceq: \beta\}$. For example $s = \lambda x.\ if\ x = 0 \wedge sin(x) = 0.0\ then\ x\ else\ x)$ where $= : \alpha \to \alpha \to \mathbb{B}$ and $sin : \mathbb{R} \to \mathbb{R}$. Constraint resolution can produce the two substitutions $\{\alpha \mapsto \mathbb{N}, \beta \mapsto \mathbb{N}\}$ and $\{\alpha \mapsto \mathbb{R}, \beta \mapsto \mathbb{R}\}$, i.e. s can receive the two types $\mathbb{N} \to \mathbb{N}$ and $\mathbb{R} \to \mathbb{R}$. A simple-minded extension of our algorithm to *let* might choose one of the two substitutions to type u and would necessarily fail. However, if we consider $u[s/f]$ instead of t, our algorithm can insert suitable coercions to make the term type correct. Unfortunately this is not a shortcoming of the hypothetical extension of our algorithm but of coercive subtyping in general: there is no way to insert coercions into t to make it type correct according to Hindley-Milner. If you want subtyping without extending the Hindley-Milner type system, there is no complete typing algorithm for *let* terms that simply inserts coercions. You may need to expand or otherwise transform *let* first.

References

1. Benke, M.: Complexity of type reconstruction in programming languages with subtyping. Ph.D. thesis, Warsaw University (1997)
2. Bourdoncle, F., Merz, S.: On the integration of functional programming, class-based object-oriented programming, and multi-methods. Research Report 26, Centre de Mathématiques Appliquées, Ecole des Mines de Paris (March 1996)
3. Coq development team: The Coq proof assistant reference manual. INRIA (2010), http://coq.inria.fr version 8.3
4. Frey, A.: Satisfying subtype inequalities in polynomial space. Theor. Comput. Sci. 277(1-2), 105–117 (2002)
5. Fuh, Y.C., Mishra, P.: Type inference with subtypes. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 94–114. Springer, Heidelberg (1988)
6. Luo, Z.: Coercions in a polymorphic type system. Mathematical Structures in Computer Science 18(4), 729–751 (2008)
7. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. 17(3), 348–375 (1978)
8. Mitchell, J.C.: Coercion and type inference. In: POPL, pp. 175–185 (1984)
9. Mitchell, J.C.: Type inference with simple subtypes. J. Funct. Program. 1(3), 245–285 (1991)
10. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Huet, G., Plotkin, G. (eds.) Logical Environments, pp. 164–188. CUP (1993)
11. Nordlander, J.: Polymorphic subtyping in O'Haskell. Sci. Comput. Program. 43(2-3), 93–127 (2002)
12. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
13. Pottier, F.: Simplifying subtyping constraints: a theory. Information & Computation 170(2), 153–183 (2001)
14. Pratt, V.R., Tiuryn, J.: Satisfiability of inequalities in a poset. Fundam. Inform. 28(1-2), 165–182 (1996)
15. Saïbi, A.: Typing algorithm in type theory with inheritance. In: POPL, pp. 292–301 (1997)
16. Shields, M., Peyton Jones, S.: Object-oriented style overloading for Haskell. In: First Workshop on Multi-language Inferastructure and Interoperability (BABEL 2001), Firenze, Italy (September 2001)
17. Simonet, V.: Type Inference with Structural Subtyping: A Faithful Formalization of an Efficient Constraint Solver. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 283–302. Springer, Heidelberg (2003)
18. Tiuryn, J.: Subtype inequalities. In: LICS, pp. 308–315 (1992)
19. Traytel, D., Berghofer, S., Nipkow, T.: Extending Hindley-Milner type inference with coercive subtyping (long version) (2011), www.in.tum.de/~nipkow/pubs/coercions.pdf
20. Wand, M., O'Keefe, P.: On the complexity of type inference with coercion. In: FPCA 1989: Functional Programming Languages and Computer Architecture, pp. 293–298. ACM, New York (1989)

Polymorphic Multi-stage Language with Control Effects

Yuichiro Kokaji and Yukiyoshi Kameyama

University of Tsukuba, Japan
kokaji@logic.cs.tsukuba.ac.jp, kameyama@acm.org

Abstract. Multi-stage programming (MSP) is a means for run-time code generation, and has been found promising in various fields including numerical computation and domain specific languages. An important problem in designing MSP languages is the dilemma of safety and expressivity; many foundational calculi have been proposed and proven to be type safe, yet, they are not expressive enough. Taha's MetaOCaml provides us a very expressive tool for MSP, yet, the corresponding theory covers its purely functional subset only.

In this paper, we propose a polymorphic multi-stage calculus with delimited-control operators. Kameyama, Kiselyov, and Shan proposed a multi-stage calculus with computation effects, but their calculus lacks polymorphism. In the presence of control effects, polymorphism in types is indispensable as all pure functions are polymorphic over answer types, and in MSP languages, polymorphism in stages is indispensable to write custom generators as library functions. We show that the proposed calculus satisfies type soundness and type inference. The former is the key to guarantee the absence of scope extrusion - open codes are never generated or executed. The latter is important in the ML-like programming languages. Following Calcagno, Moggi and Taha's work, we propose a Hindley-Milner style type inference algorithm to obtain principal types for given expressions (if they exist).

1 Introduction

Writing a code generator as a metaprogram is a vital means to achieve efficiency and maintainability simultaneously. Typed multi-stage (multi-level) programming languages help us write code generators easily and intuitively. The merit of typed multi-stage calculus over its untyped cousin, the quasiquote and unquote mechanism in Scheme, is the static assurance of type soundness: it subsumes not only type safety of code generators, but also that of generated codes, which in turn subsumes the absence of *scope extrusion*: a closed code generator never generates open codes (codes with free variables).

Many researchers have addressed the problem of assuring type soundness for multi-stage calculi; foundational calculi based on modal logic include $\lambda^{\Box}$ by Davies and Pfenning [5], $\lambda^{\bigcirc}$ by Davies [4], and $\lambda^{\bigcirc\Box}$ by Yuse and Igarashi[17]. More expressive calculi have been proposed such as λ^{α} by Taha and Nielsen [13], λ^{i} by Calcagno, Moggi and Taha [2], λ^{sim}_{open} by Kim, Yi and Calcagno [8], and the calculus by Tsukada and Igarashi [15].

Our goal is to extend the applicability of multi-stage programing so that one can write efficient code generators naturally, while keeping static type soundness. This is a challenging goal as efficient code generation (such as let-insertion) often needs impure

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 105–120, 2011.

(effectful) operations, while existing theories guarantee type soundness for purely functional subcalculi only. A recent hot topic is to add computational effects into multi-stage languages [7,10,16]. All of them remain monomorphic setting, though.

In this paper, we introduce ML-like polymorphism into multi-stage languages with computational effects, and in particular, we propose a polymorphic multi-stage calculus with delimited-control operators which satisfies type soundness. We think polymorphism is necessary in this kind of calculi by the following reasons:

- Many useful combinators for code generation are polymorphic functions[1], and, therefore, we need polymorphism to build a useful library for code generation.
- In the presence of computational effects, a type system of MSP calculi necessarily becomes so called a type-and-effect system. Then, all the pure functions (without effects) in existing libraries should be polymorphic over effects. In the case of delimited-control operators, the effects are expressed as the answer types, and, therefore, polymorphism in effects boils down to polymorphism in (answer) types.
- In MSP calculi based on Taha and Nielsen's λ^{α} or Calcagno et al.'s λ^{i}, polymorphism in environment classifiers[2] is necessary to write code generators as libraries. The staged power function is the simplest example for MSP, which already needs polymorphism in classifiers, if written as a library function (that is, not inlined).

Introducing ML-like let-polymorphism is not as trivial as one might expect. The value restriction used in ML families is too restrictive, as we want to generate polymorphic functions as the result of code generation. Other syntactic conditions do not seem suitable, either. Our solution for this problem is to revisit the semantic notion of purity, proposed by Asai and Kameyama [1] for the unstaged polymorphic calculus with delimited-control operators. A term is called pure if it does not have computational effects observable from outside. Asai and Kameyama have shown that a pure term can be made polymorphic. Following them, we allow polymorphism only for pure terms in this paper. Surprisingly, this simple idea works: it rules out all dangerous terms, while we retain the expressivity.

The proposed calculus extends Kameyama, Kiselyov, and Shan's calculus in the sense that we add let-polymorphism and the run-construct (for code execution) to their calculus. We prove type soundness of our calculus under the purity restriction, which implies that open codes are never generated or executed. We also show Hindley-Milner's style type inference algorithm for our calculus, which gives principal types if they exist.

The rest of this paper is organized as follows: Section 2 shows several example programs using multi-stage calculi and control operators, which need polymorphism. Section 3 explains the key idea of introducing polymorphism safely. Section 4 introduces our calculus λ^{DC}_{let} and operational semantics, and Section 5 introduces its type system. Then we show several useful properties such as type soundness in Section 6 and the existence of principal types in Section 7. Section 8 states concluding remarks.

[1] We will see an example of code generation combinators in Section 2.

[2] Environment classifiers are identifiers for stages, first introduced by Taha and Nielsen [13].

2 Preliminaries

This section is an example-based introduction to MSP and delimited-control operators. A comprehensive introduction to this subject may be found in the literature [11,12,7]. We use MetaOCaml to write concrete programs[3] in this section. It has three constructs for code generation: brackets, escape, and run. We do not treat CSP (cross-stage persistence) in this paper.

Staged Power Function. The first, canonical example of MSP is the staged version of the power function:

```
let rec s_power n x =
  if n = 1 then x
  else < ~x * ~(s_power (n-1) x)>
```

The expression <*e*> (bracket expression) represents a code which is not executed at the present stage, but executed at the future (next) stage. We can splice a code fragment into another code by an escape expression ∼*e*. In the expression <∼x ∗ ···>, the subexpression x is executed, and its value is spliced in this code. For instance, if we evaluate `(fun x-> <~x * 2>) <3 + 4>`, we get `<(3 + 4) * 2>`. For those familiar with Scheme macros, brackets and escape, resp, correspond to quasiquote and backquote (unquote), resp.

By executing the expression `<fun x -> ~(s_power 5 <x>)>`, we get

```
<fun x_1 -> (x_1*(x_1*(x_1*(x_1*x_1))))
```

as its value. Note that the bound variable `x` has been renamed to `x_1` during the computation, which means that variables in codes are lexically bound unlike the template mechanisms in C++ and Haskell. We can run the resulting code internally by the run construct (`!`). The computation of the expression:

```
let power5 = ! <fun x -> ~(s_power 5 <x>)>
```

yields a function equivalent to

```
fun x_1 -> (x_1*(x_1*(x_1*(x_1*x_1))))
```

which can be used at the present stage, rather than the future stage.

Let us consider the type of `s_power`. Intuitively, it has type `int` → `<int>` → `<int>` where `<int>` is the type of codes for integer expressions. Calcagno et al.'s λ^{i}_{let}, the underlying calculus of MetaOCaml, assigns to each future stage an *environment classifier* (classifier for short), in order to distinguish different next stages from each other. Hence `s_power` has type `int` → `<int>`$^{\ell}$ → `<int>`$^{\ell}$ where ℓ is a classifier. Since `s_power` is polymorphic over stages (it is not specific to any environment classifiers), its type should be polymorphic over classifiers as: $\forall \ell.$(`int` → `<int>`$^{\ell}$ → `<int>`$^{\ell}$).

Code Generation Combinators. Combinators provide us useful patterns of generating and manipulating code fragments. One of the simplest, but widely used combinators is the following `eta`:

```
let eta f = <fun x -> ~(f <x>)>
in eta (fun y -> <fun z -> z + ~y>)
```

[3] We use slightly simplified notation for multi-stage constructs: we write <*e*> for the MetaOCaml notation `.<`*e*`>.` and we suppress type variables corresponding to environment classifiers.

which, when executed, yields `<fun x -> fun z -> z + x>`. It is easy to see that `eta` should have the polymorphic type: $\forall\ell.\forall\sigma.\forall\tau.(\texttt{<}\sigma\texttt{>}^{\ell} \to \texttt{<}\tau\texttt{>}^{\ell}) \to \texttt{<}\sigma \to \tau\texttt{>}^{\ell}$.

The staged power function may be generalized to an arbitrary binary function:

```
let rec s_iterate n f x =
  if n = 1 then x
  else < ~f ~x ~(s_iterate (n-1) f x)>
in let s_iterate5 =
  eta (fun f -> eta (s_iterate 5 f))
```

which, when executed, returns `<fun f -> fun x -> f x (f x (f x (f x x)))>`. MetaOCaml assigns a monomorphic type to this expression, but we hope to assign a polymorphic type: $\forall\ell.\forall\sigma.\texttt{<}(\sigma \to \sigma \to \sigma) \to \sigma \to \sigma\texttt{>}^{\ell}$ to this expression. Here, polymorphism is necessary both at the present stage and the future stage.

Delimited-Control Operators. Control operators in functional languages are constructs for changing the order of execution, causing computational effects. Typical control operators are catch/throw (Lisp), exception (ML, Java), and call/cc (Scheme, SML/NJ). While call/cc provides an access to *unlimited* continuations, delimited-control operators provide an access to *part* of the current continuations (*delimited* continuations).

The following examples use Danvy and Filinski's delimited-control operators shift and reset [3]:

```
1 + reset (10 + 20)                          yields  31
1 + reset (10 + (shift k -> 20))             yields  21
1 + reset (10 + (shift k -> (k (k 20))))     yields  41
```

We sometimes write shift and reset, resp., as $Sk.e$ and $\{e\}$, resp.[4] `reset` denotes a delimiter, and does nothing if there is no `shift` as shown in the first line. In the second and third lines, `shift` captures the continuation (an evaluation context) up to the nearest `reset` operator, and binds the variable `k` to it. In the example, the captured continuation is `reset (10 + [ ])` where `[ ]` is a hole. It is bound to `k` and may be used later. In the third line, `k (k 20)` evaluates to `reset (10 + (reset (10 + 20)))`.

In the presence of delimited-control operators, the type system should take into account control effects (thus becomes a type-and-effect system). If we execute the following program:

```
let f = fun x -> shift k -> (k 10) + 20
in reset (f 30 >= 40)
```

the continuation captured by `shift` is `reset ([ ] >= 40)`. Then we evaluate `(reset (10 >= 40)) + 20`, which raises a run-time type error.

The computational effect caused by `shift` and `reset` can be described by an *answer type*, the return type of a delimited continuation. In the above program, the `shift` expression expects the answer type to be `int` (since the captured continuation `k` is expected to return an integer), while the `reset` expression provides `bool` as its answer

[4] In the literature, a reset expression is denoted by $\texttt{<}e\texttt{>}$. We write $\{e\}$ to avoid conflict with a bracket expression.

type. In the type-and-effect system, a function type takes the form $\sigma \to \sigma'/\beta$, where β represents the answer type. This change of type system makes all pure functions to be polymorphic over effects: for instance, the function `fun x -> x + 10` should have type $\texttt{int} \to \texttt{int}/\beta$ for any type β.

Delimited-control operators such as `shift` and `reset` have been found rather expressive: Filinski [6] proved that they can express any monadic effect, and Kiselyov, Shan and Sabry have shown a concise encoding of dynamic binding and local states in terms of them [9]. Kameyama, Kiselyov and Shan [7] have introduced control operators to type-safe multi-stage calculi, and shown that memoization in code generators is expressible as let-insertion in the calculus. In this paper, we proceed one step further, to introduce polymorphism into their calculus.

3 Introducing Polymorphism Safely

In this section, we investigate the problem of introducing let-polymorphism into a multi-stage calculus with effects, and informally show our ideas to solve the problem.

Value Restriction. As is well known, unrestricted combination of computational effects (such as states and control) and polymorphism leads to type unsoundness, and the value restriction is the standard solution for this problem: for an expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, e_1 must be a (syntactic) value for x to have a polymorphic type in e_2.

Unfortunately the value restriction is too restrictive in multi-stage calculi: in unstaged calculi we only define polymorphic functions, but in staged calculi we want to *generate* (the codes of) polymorphic functions, while value restriction prohibits runtime code generation of such polymorphic functions. Let us consider the following example:

```
let iterate5 = ! s_iterate5
in
  iterate5 (fun x y -> x * y) 3;
  iterate5 (fun x y -> x ^ y) "abc"
```

In this program snippet, the expression `! s_iterate5` is not a value, and, therefore, `iterate5` cannot have a polymorphic type under the value restriction.

The Problem. We need a better criterion as to which terms can be polymorphic. This is not a trivial problem as one might expect. To see the problem, let us consider the program `let y = e in <let x = ~y in e2>`. Then we have different situations depending on the expression `e`:

- if `e` evaluates to a code of a value, say, `<fun z-> z>`, then `x` can be polymorphic.
- if `e` evaluates to a code of an effectful computation (say, `<shift k -> 10>`), then `x` cannot be polymorphic.

In summary, it is not possible to decide the condition by simply looking at the expression e_1 in $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. In other words, the condition must be context sensitive.

Purity Restriction in Unstaged Calculus. Asai and Kameyama [1] have proposed a more liberal condition for let-polymorphism, called the purity restriction, for the (unstaged) calculus with the delimited-control operators `shift` and `reset`.

An expression is *pure* if there are no computational effects that are observable from outside. In the calculus with `shift` and `reset`, the only observable effect (other than termination) is the control effect caused by `shift`, so an expression is pure if all the calls to shift are captured within this expression, and is not pure otherwise. A pure expression is polymorphic in the answer types [14], and we can determine if a given expression is pure or not by tracking its answer type. However, type inference for such a type system is hard, and they replaced it by its conservative approximation as:

Definition 1 (Syntactic Purity [1]). *An (unstaged) expression is syntactically pure if it is a value or a reset expression* $\{e\}$.

The syntactic purity is a stronger (more restrictive) notion than purity, since a pure expression is not necessarily syntactically pure, for instance, *Sk.k* 10. However, there is no loss of expressivity by choosing syntactic purity, since, for any pure e, we can add a superfluous reset as $\{e\}$ while preserving typability and operational behavior. Asai and Kameyama have proven type soundness as well as other desirable properties for their calculus under the syntactic purity restriction.

Purity Restriction in Multi-stage Calculus. We borrow their idea to formulate the notion of (syntactic) purity in multi-stage calculi, and apply it to let-polymorphism. Since a level-0 expression (present stage expression) cannot have level-1 effects (computation effects of future stage), we formulate (semantic) purity as follows:

- A level-0 expression (an expression at the present stage) is pure if and only if it does not have observable computational effects of level-0.
- A level-1 expression (an expression at the future stage) is pure if and only if it does not have observable computational effects of level-0 and level-1.

As in the case of unstaged calculus, this semantic notion of purity is hard to decide, and we replace it by syntactic purity as follows:

- For a level-0 let-expression `let` $x = e_1$ `in` e_2, the expression e_1 must be a syntactic value or in the form $\{e_1'\}$.
- For a level-1 let-expression `let` $x = e_1$ `in` e_2, the expression e_1 must be a syntactic value or in the form $\{\sim\{\texttt{<}e_1'\texttt{>}\}\}$.

The former is the same as syntactic purity in the unstaged calculus. As for the latter, for a level-1 expression e_1', the expression $\{\sim\{\texttt{<}e_1'\texttt{>}\}\}$ introduces a level-0 reset, and then the outer most reset is of level-1. In summary, this expression has resets of both levels.

Syntactic Purity in Action. When we introduce the syntax of our calculus, we need one more twist. Rather than directly treating the (syntactically) pure expressions in the above forms, we instead use polymorphic let expression `plet` $x = e_1$ `in` e_2, which intuitively means `let` $x = \{e_1\}$ `in` e_2 for level-0, and `let` $x = \{\sim\{\texttt{<}e_1\texttt{>}\}\}$ `in` e_2 for level-1. This change of syntax greatly simplifies our formulation and it is called the *implicit-delimiter* method.

The implicit-delimiter method was also used in the literature for a different purpose; Kameyama, Kiselyov and Shan [7] regarded a level-1 binder as a delimiter for level-0, advocating that future-stage binders delimit present-stage control effects. For instance,

the level-1 expression $\lambda x.e$ is intuitively equivalent to $\lambda x.{\sim}\{\texttt{<}e\texttt{>}\}$ which has a level-0 delimiter (reset).[5]

In fact, we need both techniques in our calculus – one for ensuring syntactic purity and the other for regarding binders as delimiters. We list all the uses of implicit delimiters below.

$$
\begin{array}{lrcl}
\text{(level 0)} & \texttt{run}\, e & \equiv & \texttt{run}\, \texttt{<}\{{\sim}\{e\}\}\texttt{>} \\
& \texttt{plet}\, x = e_1\, \texttt{in}\, e_2 & \equiv & \texttt{plet}\, x = \{e_1\}\, \texttt{in}\, e_2 \\
\text{(level-1)} & \lambda x.e & \equiv & \lambda x.{\sim}\{\texttt{<}e\texttt{>}\} \\
& Sk.e & \equiv & Sk.{\sim}\{\texttt{<}e\texttt{>}\} \\
& \texttt{plet}\, x = e_1\, \texttt{in}\, e_2 & \equiv & \texttt{plet}\, x = \{{\sim}\{\texttt{<}e_1\texttt{>}\}\}\, \texttt{in}\, {\sim}\{\texttt{<}e_2\texttt{>}\}
\end{array}
$$

Note that one should understand the above equivalences (denoted by $\equiv$) as informal ones. They will help us understand some reduction rules in Section 4 and the type system in Section 5, but they are not formal entities.

Summary and Discussion. We introduce the syntactic approximation of the notion of purity in the multi-stage calculus. The notion of syntactic purity meets all our needs for let-polymorphism: it is liberal so that we can generate the codes of polymorphic functions in run-time. It is safe in the sense that type soundness holds for our calculus. It is easy to decide if a given expression is pure or not.

We believe that disallowing uncaptured calls to shift in polymorphic functions is reasonable and our implicit-delimiter approach relies on this assumption. In our experience, polymorphism and computational effects in MSP languages are completely separated, and, therefore, our purity restriction is not problematic. However, this is not at all a final word, and a further study is left for future work.

4 The Calculus

This section introduces the polymorphic multi-stage calculus λ_{let}^{DC}, which is based on λ^{i} by Calcagno et al., and $\lambda_{1}^{\oslash}$ by Kameyama et al. The former has polymorphism, more than two levels, CSP, but no control operators. The latter has control operators but no polymorphism, no `run` constructs and no CSP, and is restricted to two levels. Our calculus λ_{let}^{DC} has control operators, polymorphism, `run` constructs, but currently does not have CSP, and restricted to two levels. The principles of our design are simplicity and essence: the combination of control operators, polymorphism, and `run` constructs are the usual sources of type unsoundness, thus leading to scope extrusion, while adding more than two stages and CSP seems orthogonal to these problems. It is desirable to have all these features, but in this paper, to avoid clutter, we present a minimal calculus which exposes the subtle problems in the design of multi-stage calculi. Extension to more expressive calculi are left for future work.

We restrict the computational effects to those caused by `shift` and `reset`, and their answer types be invariant through the computation (no answer-type modification). Danvy's type-safe `printf` is a typical example which needs the effect of answer-type

[5] We inserted brackets and escape to make the reset be of level 1.

$$e^0 ::= v^0 \mid e^0 e^0 \mid e^0 + e^0 \mid \texttt{plet}\ x = e^0\ \texttt{in}\ e^0 \mid \{e^0\} \mid Sk.e^0 \mid \texttt{<}e^1\texttt{>} \mid \texttt{run}\ e^0$$
$$e^1 ::= v^1 \mid e^1 e^1 \mid e^1 + e^1 \mid \texttt{plet}\ x = e^1\ \texttt{in}\ e^1 \mid \{e^1\} \mid Sk.e^1 \mid {\sim}e^0$$
$$v^0 ::= x \mid i \mid \lambda x.e^0 \mid \texttt{<}v^1\texttt{>}$$
$$v^1 ::= x \mid i \mid \lambda x.v^1 \mid v^1 v^1 \mid v^1 + v^1 \mid \texttt{plet}\ x = v^1\ \texttt{in}\ v^1 \mid \{v^1\} \mid Sk.v^1$$

Fig. 1. Syntax of Expressions

modification if written in direct style, so it is not typable in our calculus. However, this restriction is only for presentation; we can develop the calculus with answer-type modification, although it doubles the number of answer types in the judgment. We believe that our calculus provides a useful information on how to introduce polymorphism into multi-stage calculus with computational effects.

Syntax. We define the syntax of λ_{let}^{DC}. We assume to have an infinite number of environment classifiers (or classifiers) $\ell, \ell_1, \ell_2, \cdots$. They are abstract entities; variables for classifiers are quantified by $\forall$, but there are no constants for classifiers. The stage-level L is either 0 (for the present stage), or a single classifier ℓ (for the next, or future stage). In general a level is a finite sequence of classifiers, but we restrict the number of levels to two, so the maximum length of levels is 1. When the names of classifiers do not matter, all stage-levels ℓ_i are simply called "level 1".

Fig. 1 defines the type-free expressions where e^n and v^n, resp., are a level-n expression and a level-n value, resp., for $n = 0, 1$.

A level-0 expression e^0 is either a variable x, an integer literal i, λ-abstraction $\lambda x.e^0$, addition $e^0 + e^0$, a polymorphic let expression $\texttt{plet}\ x = e^0\ \texttt{in}\ e^0$, a reset expression $\{e^0\}$, a shift expression $Sk.e^0$, a bracket expression $\texttt{<}e^1\texttt{>}$, or a run expression $\texttt{run}\ e^0$. Note that, a level-1 expression e^1 should come inside a bracket expression.

A level-1 expression e^1 contains an escape expression ${\sim}e^0$, but since the maximum level is one, there are no expressions like $\texttt{<}e^2\texttt{>}$ or $\texttt{run}\ e^1$.

A level-0 value v^0 is standard except that a bracket expression $\texttt{<}v^1\texttt{>}$ constitutes a code value. Note that we will introduce call-by-value operational semantics. The definition of a level-1 value v^1 contains all kinds of expressions except an escape expression.

The variable x in $\lambda x.e$ and $\texttt{plet}\ x = e'\ \texttt{in}\ e$, k in $Sk.e$ are bound in each e. We identify α-equivalent expressions as usual, and $\texttt{FV}(e)$ denotes the set of free variables in e. Given an expression e, a variable x and a value v of the same level as x, $e[v/x]$ denotes the result of substitution of v for x in e.

Operational Semantics. We define the call-by-value operational semantics. Fig. 2 defines evaluation contexts of various levels. E^{ij} denotes an evaluation context such that its hole (denoted by $\bullet$) will be filled by a level-j expression, and then the whole context will become a level-i expression. An interesting one is $E^{01}[\lambda x.\bullet]$, which means that we evaluate under lambda abstraction.

We also define a pure evaluation context F^{0j} for $j = 0, 1$. Intuitively, this context does not have resets which enclose the hole. In our calculus, certain expressions have implicit resets, and they cannot constitute pure evaluation contexts.

$$
\begin{aligned}
E^{00} ::= {} & \bullet \mid E^{00}[\bullet e^0] \mid E^{00}[v^0 \bullet] \mid E^{00}[\bullet + e^0] \mid E^{00}[v^0 + \bullet] \\
& \mid E^{00}[\texttt{plet}\ x = \bullet\ \texttt{in}\ e^0] \mid E^{00}[\{\bullet\}] \mid E^{00}[\texttt{run}\ \bullet] \mid E^{01}[\sim\bullet] \\
E^{01} ::= {} & E^{01}[\bullet e^1] \mid E^{01}[v^1 \bullet] \mid E^{01}[\bullet + e^1] \mid E^{01}[v^1 + \bullet] \mid E^{01}[\lambda x.\bullet] \\
& \mid E^{01}[\texttt{plet}\ x = \bullet\ \texttt{in}\ e^1] \mid E^{01}[\texttt{plet}\ x = v^1\ \texttt{in}\ \bullet] \\
& \mid E^{01}[Sk.\bullet] \mid E^{01}[\{\bullet\}] \mid E^{00}[<\bullet>] \\
F^{00} ::= {} & \bullet \mid F^{00}[\bullet e^0] \mid F^{00}[v^0 \bullet] \mid F^{00}[\bullet + e^0] \mid F^{00}[v^0 + \bullet] \mid F^{01}[\sim\bullet] \\
F^{01} ::= {} & F^{01}[\bullet e^1] \mid F^{01}[v^1 \bullet] \mid F^{01}[\bullet + e^1] \mid F^{01}[v^1 + \bullet] \mid F^{00}[<\bullet>]
\end{aligned}
$$

Fig. 2. Evaluation Contexts

$$
\begin{aligned}
E^{00}[i+j] &\rightsquigarrow E^{00}[m] \ \text{ if } \ i+j=m & \qquad E^{00}[\{v^0\}] &\rightsquigarrow E^{00}[v^0] \\
E^{00}[(\lambda x.e^0)v^0] &\rightsquigarrow E^{00}[e^0[v^0/x]] & E^{01}[\sim<v^1>] &\rightsquigarrow E^{01}[v^1] \\
E^{00}[\texttt{plet}\ x = v^0\ \texttt{in}\ e^0] &\rightsquigarrow E^{00}[e^0[v^0/x]] & E^{00}[\texttt{run}\ <v^1>] &\rightsquigarrow E^{00}[v^1] \\
E^{00}[\{F^{00}[Sk.e^0]\}] &\rightsquigarrow E^{00}[\{e^0[\lambda x.\{F^{00}[x]\}/k]\}]
\end{aligned}
$$

Fig. 3. Reduction Rules

Fig. 3 gives the reduction rules in the evaluation-context style.

The first three rules are integer addition, β reduction in call-by-value, and let-reduction as usual. The next two rules are the ones for control operators. shift captures the continuation delimited by the nearest delimiter. In the rule, F^{00} is an evaluation context which does not have resets around the hole, which means that the reset displayed in the rule is the nearest one. After capturing the delimited context $\{F^{00}\}$, we convert it to a functional form $\lambda x.\{F^{00}[x]\}$, and bind k to it, and continue the evaluation. For the next rule, if the body of a reset expression is a level-0 value, the delimiter is simply discarded.

The last two rules are the reduction rules for multi-stage constructs. In the second last rule, the evaluation context E^{01} signifies that the hole in E^{01} is of level-1, which means that there are brackets enclosing the hole. Hence the subexpression $\sim<v^1>$ appears in a code, and thus we are splicing the code v^1 into the code. Then it is easy to understand the reduction rule. The last rule defines the code execution. If the body of the run expression is $<v^1>$, then we extract the content v^1 of the code expression, and start evaluating v^1. In the right-hand side of this rule, a level-1 value v^1 is plugged in to the level-0 hole in E^{00}.

Reductions for Implicit Delimiter. One may notice that reduction rules Fig. 3 are too weak. In fact, there are closed, non-value expressions that may not be reduced by any reduction rules. For instance, $\{<\lambda x.\sim(Sk.e)>\}$ gets stuck, since the obvious candidate for reduction is $E^{00}[\{F^{00}[Sk.e^0]\}] \rightsquigarrow \cdots$, but the definition of F^{00} does not allow level-1 abstraction $\lambda x.e$. Our calculus rules out some of such kinds of expressions, but not all of them. Some expressions in the above form typecheck in our type system, and thus, we need additional reduction rules for those safe patterns, to take into account the implicit delimiters.

Auxiliary definition for pure evaluation contexts:

$$F^{10} ::= F^{10}[\bullet e^0] \mid F^{10}[v^0 \bullet] \mid F^{10}[\bullet + e^0] \mid F^{10}[v^0 + \bullet] \mid F^{11}[\sim \bullet]$$
$$F^{11} ::= \bullet \mid F^{11}[\bullet e^1] \mid F^{11}[v^1 \bullet] \mid F^{11}[\bullet + e^1] \mid F^{11}[v^1 + \bullet] \mid F^{10}[<\bullet>]$$

Additional reduction for level-0 implicit resets:

$$G[F^{00}[Sk.e^0]] \rightsquigarrow G[\{e^0[\lambda x.\{F^{00}[x]\}/k]\}] \qquad \text{where } G ::= E^{00}[\texttt{plet}\ x = \bullet\ \texttt{in}\ e^0] \mid E^{00}[\texttt{run}\ \bullet]$$

Additional reduction for level-1 implicit resets:

$$H[F^{10}[Sk.e^0]] \rightsquigarrow H[\sim\{e^0[\lambda x.\{<F^{10}[x]>\}/k]\}]$$
$$\text{where } H ::= E^{01}[\lambda x.\bullet] \mid E^{01}[Sk'.\bullet] \mid E^{01}[\texttt{plet}\ x = \bullet\ \texttt{in}\ e^1] \mid E^{01}[\texttt{plet}\ x = e^1\ \texttt{in}\ \bullet]$$

Fig. 4. Reduction rules for implicit resets

Fig. 4 gives the reduction rules corresponding to implicit resets.

Although the reduction rules look complicated, they are in fact simply derived from the informal reading for implicit resets, stated earlier.

5 Type System

In this section, we define a polymorphic type system for the calculus λ^{DC}_{let}. We first define types:

$$\sigma, \tau, \alpha, \beta ::= t \mid \texttt{int} \mid \sigma \to \tau/\beta \mid <\sigma/\beta>^{\ell} \qquad \text{monomorphic type}$$
$$T ::= \sigma \mid \forall t.T \mid \forall \ell.T \qquad \text{polymorphic type}$$

where t is a type variable, and `int` is the type for integers. The type $\sigma \to \tau/\beta$ is the function type with effects, which are determined by the answer type β. The type $<\sigma/\beta>^{\ell}$ is the type for codes of level ℓ where σ is the type of the code, and β is a level-1 answer type.

The polymorphic type T is a monomorphic type with universal quantification. Following Calcagno et al. [2], we have two kinds of quantification: $\forall t.T$ represents universal quantification over types, and $\forall \ell.T$ universal quantification over environment classifiers. We sometimes write $\forall \overline{t}.\forall \overline{\ell}.\sigma$ for the type σ quantified over sequences of type variables $t_1, \cdots, t_n$ and environment classifiers $\ell_1, \cdots, \ell_m$.

For a type T, $\texttt{FC}(T)$ and $\texttt{FTV}(T)$, resp., are the set of free classifiers in T, and the set of free type variables in T, resp. In the following, we sometimes write $\texttt{FV}(\Gamma)$ and so on, which has obvious meaning.

A general form of a judgment is $\Gamma \vdash^L e : \sigma \ ; \beta_0 \ ; \beta_1$ where the type context Γ is a (possibly empty) finite sequence of the form $(x : T)^L$ where T is a polymorphic type and L is a level. The level L in $(x : T)^L$ means that the variable x can be used in level L. The above judgment means that, under the type context Γ, e is a level-L expression

of type σ with the level-0 answer type β_0 and the level-1 answer type β_1. When $L = 0$, the level-1 answer type β_1 is not significant (a level-0 expression cannot contain level-1 effects), and therefore we often write $-$ for β_1.

Typing rules of λ^{DC}_{let} are defined as follows.

$$\frac{(i \text{ is an integer constant})}{\Gamma \vdash^L i : \mathtt{int}\ ;\beta_0\ ;\beta_1}\ \text{int} \qquad \frac{(\tau\ \leq T)}{\Gamma, (x:T)^L \vdash^L x : \tau\ ;\beta_0\ ;\beta_1}\ \text{var}$$

$$\frac{\Gamma \vdash^L e_1 : \mathtt{int}\ ;\beta_0\ ;\beta_1 \quad \Gamma \vdash^L e_2 : \mathtt{int}\ ;\beta_0\ ;\beta_1}{\Gamma \vdash^L e_1 + e_2 : \mathtt{int}\ ;\beta_0\ ;\beta_1}\ \text{plus}$$

$$\frac{\Gamma, (x:\sigma)^0 \vdash^0 e : \tau\ ;\beta_0\ ;-}{\Gamma \vdash^0 \lambda x.e : \sigma \to \tau/\beta_0\ ;\alpha_0\ ;-}\ \lambda^0 \qquad \frac{\Gamma, (x:\sigma)^\ell \vdash^\ell e : \tau\ ;<\tau/\beta_1>\ ;\beta_1}{\Gamma \vdash^\ell \lambda x.e : \sigma \to \tau/\beta_1\ ;\alpha_0\ ;\alpha_1}\ \lambda^1$$

$$\frac{\Gamma \vdash^0 e_1 : \sigma \to \tau/\alpha_0\ ;\alpha_0\ ;- \qquad \Gamma \vdash^0 e_2 : \sigma\ ;\alpha_0\ ;-}{\Gamma \vdash^0 e_1 e_2 : \tau\ ;\alpha_0\ ;-}\ \text{app}^0 \qquad \frac{\Gamma \vdash^\ell e_1 : \sigma \to \tau/\alpha_1\ ;\alpha_0\ ;\alpha_1 \qquad \Gamma \vdash^\ell e_2 : \sigma\ ;\alpha_0\ ;\alpha_1}{\Gamma \vdash^\ell e_1 e_2 : \tau\ ;\alpha_0\ ;\alpha_1}\ \text{app}^1$$

$$\frac{\Gamma \vdash^0 e_1 : \sigma\ ;\sigma\ ;- \quad \overline{t} \subseteq \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma), \overline{\ell} \subseteq \mathtt{FC}(\sigma) - \mathtt{FC}(\Gamma) \quad \Gamma, (x:\forall \overline{t}.\forall \overline{\ell}.\sigma)^0 \vdash^0 e_2 : \tau\ ;\beta_0\ ;-}{\Gamma \vdash^0 \mathtt{plet}\ x = e_1\ \mathtt{in}\ e_2 : \tau\ ;\beta_0\ ;-}\ \text{let}^0$$

$$\frac{\Gamma \vdash^\ell e_1 : \sigma\ ;<\sigma/\sigma>^\ell\ ;\sigma \quad \overline{t} \subseteq \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma) \quad \Gamma, (x:\forall \overline{t}.\sigma)^\ell \vdash^\ell e_2 : \tau\ ;<\tau/\beta_1>^\ell\ ;\beta_1}{\Gamma \vdash^\ell \mathtt{plet}\ x = e_1\ \mathtt{in}\ e_2 : \tau\ ;<\tau/\beta_1>^\ell\ ;\beta_1}\ \text{let}^1$$

$$\frac{\Gamma, (k:\forall t.(\sigma \to \beta_0/t))^0 \vdash^0 e : \beta_0\ ;\beta_0\ ;-}{\Gamma \vdash^0 Sk.e : \sigma\ ;\beta_0\ ;-}\ \text{shift}^0 \qquad \frac{\Gamma \vdash^0 e : \beta_0\ ;\beta_0\ ;-}{\Gamma \vdash^0 \{e\} : \beta_0\ ;\alpha_0\ ;-}\ \text{reset}^0$$

$$\frac{\Gamma, (k:\forall t.(\sigma \to \beta_1/t))^\ell \vdash^\ell e : \beta_1\ ;<\beta_1/\beta_1>^\ell\ ;\beta_1}{\Gamma \vdash^\ell Sk.e : \sigma\ ;\alpha_0\ ;\beta_1}\ \text{shift}^1 \qquad \frac{\Gamma \vdash^\ell e : \beta_1\ ;\beta_0\ ;\beta_1}{\Gamma \vdash^\ell \{e\} : \beta_1\ ;\beta_0\ ;\alpha_1}\ \text{reset}^1$$

$$\frac{\Gamma \vdash^\ell e : \sigma\ ;\beta_0\ ;\beta_1}{\Gamma \vdash^0 <e> : <\sigma/\beta_1>^\ell\ ;\beta_0\ ;-}\ \text{brackets}^0 \qquad \frac{\Gamma \vdash^0 e : <\sigma/\beta_1>^\ell\ ;\beta_0\ ;-}{\Gamma \vdash^\ell \sim e : \sigma\ ;\beta_0\ ;\beta_1}\ \text{escape}^1$$

$$\frac{\Gamma \vdash^0 e : <\sigma/\sigma>^\ell\ ;<\sigma/\sigma>^\ell\ ;- \quad (\ell \notin \mathtt{FC}(\Gamma, \sigma))}{\Gamma \vdash^0 \mathtt{run}\ e : \sigma\ ;\alpha_0\ ;-}\ \text{run}^0$$

Let us explain the typing rules briefly.

The var rule is the standard one in polymorphic type systems where $\tau \leq T$ means that τ is an instance of polymorphic type T. More precisely, if $T = \forall \overline{t}.\forall \overline{\ell}.\sigma$ for a monomorphic type σ, then $\tau = \sigma[\overline{\alpha}/\overline{t}][\overline{\ell'}/\overline{\ell}]$ for some $\alpha_1, \cdots, \alpha_n, \ell'_1, \cdots, \ell'_k$.

The λ^0 rule is also standard except that it retrieves the level-0 answer type β_0 into the function type $\sigma \to \tau/\beta_0$. Since a function does not have computational effects, its level-0 answer type can be an arbitrary type α_0. The λ^1 rule is the key to avoid scope extrusion as studied by Kameyama, Kiselyov and Shan [7]. As we explained in Section 3, the level-1 expression $\lambda x.e$ has an implicit level-0 reset beneath this lambda, namely, its intuitive meaning is $\lambda x.{\sim}\{\texttt{<}e\texttt{>}\}$. In order to type this expression, we need to require the level-0 answer type for e be $\texttt{<}\tau/\beta_1\texttt{>}^\ell$. The app rules can be understood easily.

The level-0 polymorphic let expression $\texttt{plet}\ x = e_1\ \texttt{in}\ e_2$ should be understood as $\texttt{let}\ x = \{e_1\}\ \texttt{in}\ e_2$, hence the answer type of e_1 must be the same as the type of e_1 itself. (See also the reset rule below.) The level-1 polymorphic let expression $\texttt{plet}\, x = e_1\ \texttt{in}\ e_2$ is slightly more complex. First, due to the purity restriction for e_1, it is understood as $\texttt{let}\, x = \{{\sim}\{\texttt{<}e_1\texttt{>}\}\}\ \texttt{in}\ e_2$. But since this level-1 let expression is a binder for e_2, it must not have level-0 effects, and therefore, $\texttt{let}\ x = \{{\sim}\{\texttt{<}e_1\texttt{>}\}\}\ \texttt{in}\ {\sim}\{\texttt{<}e_2\texttt{>}\}$ is the final meaning of polymorphic let expression. The type rule reflects this reading.

The shift rules are adaptation from the standard typing rules for them in the literature of delimited continuations. For instance, level-0 shift captures a delimited context whose answer type is β_0. Thus the type of k must be a function type whose return type is β_0. Since the delimited continuation is a pure function (no control effects are involved), it is polymorphic in the answer type, and thus we quantify t in $\sigma \to \beta_0/t$. The shift^1 rule is more complex, as it binds level-1 variables k, and again we have an implicit level-0 reset. (Note the level-0 answer type for e is $\texttt{<}\beta_1/\beta_1\texttt{>}^\ell$.)

The reset rules are also adaptation of the type rule in the literature. For a level-0 expression $\{e\}$, its type must be the same type as that of e and also the level-0 answer type of e. Since $\{e\}$ has no observable control effects, its level-0 answer type can be arbitrary type α_0. The level-1 reset rule can be understood similarly.

The rules for brackets and escapes are the same as those in λ^α and λ^i except that we need to memoize the level-0 effect (β_1) in the code type as $\texttt{<}\sigma/\beta_1\texttt{>}^\ell$. Brackets turn a level-1 expression to a level-0 expression, and escape does the converse.

Finally, the run rule is one of the most interesting ones. The run construct is to execute the code (inside brackets) at the present stage, and thus it is important to ensure it is a closed code. The calculi λ^α and λ^i ensure this closedness condition in terms of the eigen-variable condition for the classifier ℓ: it must not appear at any other place in the judgment for e. If the condition is met, e does not depend on the stage ℓ, so we can run (and compile) it at the present stage. In addition to this condition, we need to rule out, for instance, an expression like `run <shift k -> ..>`. In general we need to ensure level-0 and level-1 purity of e in this rule. Thus we implicitly introduce resets of both levels.

The following figure shows a type derivation for $\lambda x.{\sim}(Sk.\texttt{<}x + 10\texttt{>})$ where $\Gamma = (x : \texttt{int})^\ell, (k : \forall t.\texttt{<int/int>}^\ell \to \texttt{<int/int>}^\ell/t)^0$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash^{\ell} x+10 : \mathtt{int}\ ;\mathtt{<int/int>}^{\ell}\ ;\mathtt{int}}{\Gamma \vdash^{0} \mathtt{<}x+10\mathtt{>} : \mathtt{<int/int>}^{\ell}\ ;\mathtt{<int/int>}^{\ell}\ ;-}}{(x:\mathtt{int})^{\ell} \vdash^{0} Sk.\mathtt{<}x+10\mathtt{>} : \mathtt{<int/int>}^{\ell}\ ;\mathtt{<int/int>}^{\ell}\ ;-}}{(x:\mathtt{int})^{\ell} \vdash^{\ell} {\sim}(Sk.\mathtt{<}x+10\mathtt{>}) : \mathtt{int}\ ;\mathtt{<int/int>}^{\ell}\ ;\mathtt{int}}}{\vdash^{\ell} \lambda x.{\sim}(Sk.\mathtt{<}x+10\mathtt{>}) : (\mathtt{int} \to \mathtt{int/int})\ ;\beta_1\ ;\beta_2}$$

6 Type Soundness

In this section we prove type soundness of λ^{DC}_{let}, which consists of the subject reduction property (type preservation) and the progress property. Due to lack of space, we do not give complete proofs, and, instead state important lemmas in this paper.

Lemma 1 (Values do not have effects). *If $\Gamma \vdash^{0} v^0 : \sigma\ ;\beta_0\ ;-$ is derivable, then $\Gamma \vdash^{0} v^0 : \sigma\ ;\alpha\ ;-$ is derivable for any α. If $\Gamma \vdash^{\ell} v^1 : \sigma\ ;\beta_0\ ;\beta_1$ is derivable, then $\Gamma \vdash^{\ell} v^1 : \sigma\ ;\alpha\ ;\beta_1$ is derivable for any α.*

This lemma can be proven immediately. We then state two lemmas about substitution, which are used in the proof of subject reduction.

Lemma 2 (Substitution for monomorphic variable). *If $\Gamma_1 \vdash^{0} v : \sigma\ ;\alpha_0\ ;-$ and $\Gamma_1,\Gamma_2,(x:\sigma)^0 \vdash^{L} e : \tau\ ;\beta_0\ ;\beta_1$ are derivable, $\Gamma_1,\Gamma_2 \vdash^{L} e[v/x] : \tau\ ;\beta_0\ ;\beta_1$ is derivable.*

Lemma 3 (Substitution for polymorphic variable). *If $\Gamma_1 \vdash^{0} v : \sigma;\ [\,]$ and $\Gamma_1,\Gamma_2,(x : \forall t.\forall \ell.\sigma)^0 \vdash^{L} e : \tau\ ;\beta_0\ ;\beta_1$ are derivable, $t_1,\cdots,t_n \in \mathit{FTV}(\sigma) - \mathit{FTV}(\Gamma_1)$ and $\ell_1,\cdots,\ell_k \in \mathit{FC}(\sigma) - \mathit{FC}(\Gamma)$, then $\Gamma_1,\Gamma_2 \vdash^{L} e[v/x] : \tau\ ;\beta_0\ ;\beta_1$ is derivable.*

These lemmas can be proven by structural induction on the second derivation, resp. The next lemma is necessary to prove subject reduction for the case of the run construct. First, we introduce an auxiliary definition $\_\Downarrow_{\ell}$ for elements of typing contexts, defined as:

$$(x:\sigma)^0 \Downarrow_{\ell} \stackrel{\text{def}}{=} (x:\sigma)^0$$
$$(x:\upsilon)^{\ell} \Downarrow_{\ell} \stackrel{\text{def}}{=} (x:\upsilon)^0$$
$$(x:\upsilon)^{\ell'} \Downarrow_{\ell} \stackrel{\text{def}}{=} (x:\upsilon)^{\ell'} \quad \text{if } \ell' \neq \ell$$

The definition extends to $\Gamma \Downarrow_{\ell}$ straightforwardly.

Lemma 4. *Suppose $\Gamma_1,\Gamma_2 \vdash^{\ell} v^1 : \sigma\ ;\beta_0\ ;\beta_1$ is derivable such that $\ell \notin \mathit{FC}(\Gamma_1,\sigma,\beta_0,\beta_1)$, and Γ_2 consists of the form $(x_i:\tau_i)^{\ell}$ such that $\ell \notin \mathit{FC}(\tau_i)$. Then we can derive $\Gamma_1,\Gamma_2\Downarrow_{\ell} \vdash^{0} v^1 : \sigma\ ;\beta_1\ ;-$.*

This lemma is proven by induction on the derivation of $\Gamma_1,\Gamma_2 \vdash^{\ell} v^1 : \sigma\ ;\beta_0\ ;\beta_1$. As its corollary, we obtain:

Corollary 1. *Suppose* $\Gamma \vdash^{\ell} v^1 : \sigma \ ;\beta_0 \ ;\beta_1$ *is derivable such that* $\ell \notin FV(\Gamma, \sigma, \beta_0, \beta_1)$*, then we can derive* $\Gamma \vdash^0 v^1 : \sigma \ ;\beta_1 \ ;-$.

Finally, we state the subject reduction property.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash^0 e : \tau \ ;\beta_0 \ ;-$ *is derivable and* $e \rightsquigarrow^* e'$*, then* $\Gamma \vdash^0 e' : \tau \ ;\beta_0 \ ;-$ *is derivable.*

Proof. (very brief sketch) We prove the theorem by induction on the number of reduction steps, and case analysis of reductions.

For the reduction rules of shift (when shift captures a continuation up to the nearest reset), we have the important observation: for any type derivation of a pure context F^{ij} $(i, j = 0, 1)$, there are no (explicit or implicit) level-0 resets and no level-0 binders which enclose the hole. Moreover, there are no binders of the same level. Then by induction on F^{ij}, we can prove that the level-0 answer type does not change through this derivation. By using this fact, we can prove the subject reduction property for this case.

The cases for β and let reductions are handled by Lemmas 2 and 3. Other cases are proven straightforwardly.

The progress property states that a closed well-typed express does not get stuck.

Theorem 2 (Progress). *If* $\vdash^0 e : \tau \ ;\tau \ ;-$ *is derivable, there exists an expression* e_1 *such that* $\{e\} \rightsquigarrow e_1$.

Proof. (sketch) We first prove by induction that, for any typable expression e, it is a value, a reducible expression, or a stuck expression $E^{00}[Sk.e']$.

Then, we can prove that $\{e\}$ is always a redex.

7 Principal Type and Type Inference

Type inference is an important feature for ML-like languages, and it is even more important for our calculus, since we need to keep track of the effects of an expression as type annotation. In this section we briefly mention the type inference algorithm for λ_{let}^{DC}.

Calcagno, Moggi and Taha [2] proposed the calculus λ^i and its polymorphic version λ_{let}^i, which is slightly less expressive than the earlier calculus λ^α by Taha and Nielsen, but has principal typing (or principal type for λ_{let}^i). λ^i is suitable as the foundation of multi-stage programming languages, as demonstrated by the success of the MetaOCaml language.

Our calculus λ_{let}^{DC} also has the principal type property, and a sound and complete type inference algorithm similar to the algorithm W.

Given an expression e, a type context Γ and a level L, we say $(\theta, \sigma, \alpha, \beta)$ is a solution for these data if and only if θ is a substitution for type variables and classifiers, σ, α, β are types, and $\Gamma \vdash^L e : \sigma \ ;\alpha \ ;\beta$ is derivable.

Proposition 1 (Principal Type). *Suppose there is a solution for an expression e, a type context* Γ*, and a level L. Then there exists a principal solution* $(\theta_0, \sigma_0, \alpha_0, \beta_0)$ *for them, namely, for any solution* $(\theta_1, \sigma_1, \alpha_1, \beta_1)$ *for the same input, there exists a substitution* ϕ *such that* $\theta_1 = \theta_0\phi$*,* $\sigma_1 = \sigma_0\phi$*,* $\alpha_1 = \alpha_0\phi$ *and* $\beta_1 = \beta_0\phi$.

Here we slightly generalized the statement of the theorem than the standard form so that the inductive proof goes through.

Proof. (sketch) We can construct a Hindley-Milner's style type inference algorithm for λ_{let}^{DC}. This is due to the "implicit reset" approach, since we no longer have to infer the (semantic) purity in the process of type inference.

The only difficulty in constructing the algorithm is the case for the run rule, which has the negative side condition $\ell \notin \mathtt{FV}(\Gamma, \sigma)$. To see how our algorithm handle this case, let us consider the expression `run` e under the context Γ and the level L. We first infer the type of e under Γ and L. Suppose we get $(\theta_0, \sigma_0, \alpha_0, \beta_0)$ as the answer. Then we unify σ_0 with $<\sigma'/\sigma'>^{\ell}$ for a fresh σ' and so on. We check if the classifier ℓ appears in the results or not. The algorithm fails to infer a type if ℓ appears in a wrong place of the results, and continues otherwise. In the latter case, the future process of type inference does not unify ℓ to other classifiers, so the side condition $\ell \notin \mathtt{FV}(\Gamma, \sigma)$ will remain valid.

Hence we can build a W-like type inference algorithm. The proof of soundness and completeness of the algorithm is as standard.

8 Conclusion

We have designed a polymorphic type system for multi-stage calculus with delimited-control operators where polymorphism in types and that in classifiers are expressible. The calculus in this paper extends Kameyama, Kiselyov and Shan's calculus in that (besides polymorphism) we have the run construct. The key idea of integrating these conflicting concepts into one calculus is to relax the value restriction to the syntactic purity restriction, and then introduce the notion of implicit resets. We have proven type soundness and the existence of principal types, both of which are essential to make our language usable.

The success of this combination strongly owes to the local nature of computational effects by shift and reset: we can represent, in particular, mutable variables in terms of shift and reset, but their scope is local to a certain block in a program. By placing a sufficiently many resets (on the borders of polymorphism and run), we have obtained a type-safe polymorphic calculus.

We mention other approaches to design type-safe multi-stage calculi with control effects. Kim, Yi and Calcagno [8] proposed a polymorphic modal type system for Lisp-like languages. Since their calculus has quite different flavors in nature (for instance, α-equivalence is not admissible in their calculus, while it is built in our calculus), it is left for future work to compare these two different lines of works.

Recently Westbrook et al. [16] designed a multi-stage programming language Mint as an extension of Java. To ensure type safety, they introduced the notion of weak separability. Despite the difference of underlying languages, it will be interesting to compare their conditions with ours so that we can build an even more powerful, and type-safe calculus.

Acknowledgments. We would like to thank Kenichi Asai, Atsushi Igarashi, Oleg Kiselyov, Chung-chieh Shan, and Kwangkeun Yi. We also thank anonymous reviewers for constructive comments. The second author is supported in part by JSPS Grant-in-Aid for Scientific Research (B) 21300005.

References

1. Asai, K., Kameyama, Y.: Polymorphic Delimited Continuations. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 239–254. Springer, Heidelberg (2007)
2. Calcagno, C., Moggi, E., Taha, W.: Ml-Like Inference for Classifiers. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 79–93. Springer, Heidelberg (2004)
3. Danvy, O., Filinski, A.: Abstracting control. In: LISP and Functional Programming, pp. 151–160 (1990)
4. Davies, R.: A temporal logic approach to binding-time analysis. In: LICS, pp. 184–195 (1996)
5. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM 48(3), 555–604 (2001)
6. Filinski, A.: Representing Monads. In: Proc. 21st Symposium on Principles of Programming Languages, pp. 446–457 (1994)
7. Kameyama, Y., Kiselyov, O., Shan, C.-c.: Shifting the stage: staging with delimited control. In: PEPM, pp. 111–120 (2009)
8. Kim, I.-S., Yi, K., Calcagno, C.: A polymorphic modal type system for lisp-like multi-staged languages. In: POPL, pp. 257–268 (2006)
9. Kiselyov, O., Shan, C.-c., Sabry, A.: Delimited dynamic binding. In: Reppy, J.H., Lawall, J.L. (eds.) ICFP, pp. 26–37. ACM (2006)
10. Sugiura, K., Kameyama, Y.: Multi-stage language with control effect and code execution. Computer Software Journal of Japan Society of Software Science and Technology 28(1), 217–229 (2011) (in Japanese)
11. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation, pp. 30–50 (2003)
12. Taha, W.: A Gentle Introduction to Multi-stage Programming, Part II. In: GTTSE, pp. 260–290 (2007)
13. Taha, W., Nielsen, M.F.: Environment classifiers. In: POPL, pp. 26–37 (2003)
14. Thielecke, H.: From control effects to typed continuation passing. In: POPL, pp. 139–149. ACM Press (2003)
15. Tsukada, T., Igarashi, A.: A logical foundation for environment classifiers. Logical Methods in Computer Science 6(4/9), 1–43 (2010)
16. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: PLDI, pp. 400–411 (2010)
17. Yuse, Y., Igarashi, A.: A modal type system for multi-level generating extensions with persistent code. In: PPDP, pp. 201–212 (2006)

Compiler Backend Generation for Application Specific Instruction Set Processors

Zhen Cao, Yuan Dong, and Shengyuan Wang

Department of Computer Science and Technology,
Tsinghua University, Beijing 100084, China
cao-z08@mails.tsinghua.edu.cn,{dongyuan,wwssyy}@tsinghua.edu.cn

Abstract. Application Specific Instruction Set Processors (ASIPs) have become popular in the development of embedded systems. For these processors easily-retargetable, high-performance compilers play a key role in the development process, improving productivity and reducing time-to-market. We propose a novel, object-based architecture description language (ADL) *OpenDL,* as well as a well-structured Rule Library to automatically retarget compiler backends. OpenDL is a succinct and high-quality ADL with object-based inheritance features, while the Rule Library applies instruction templating in order to allow detailed instruction specification to handle complex rule patterns. We use these tools to automatically retarget the open source industrial-strength compiler Open64 to the high-performance embedded processor PowerPC. A reliable version of auto-retargetable industrial-strength compiler is generated which achieves comparable performance to gcc 4.5 for both the EEMBC and SPEC CPU 2000 benchmarks.

Keywords: ADL, automatic retarget, code generator generator.

1 Introduction

Power and energy consumption are ever-growing concerns, especially for embedded domains, and this has resulted in custom instruction processors becoming a trend. Correspondingly, application-specific instruction set processors (ASIPs) have become more and more popular [17] for the development of embedded applications and systems.

ASIPs put forward new challenges to the implementation of compiler retargeting since they greatly increase the number of back-end targets. For these processors, customizing dedicated optimal instruction sets play an important role in performance. To produce high performance applications, the designers need powerful development tools such as compilers, assemblers and simulators to build efficient and low energy processors. In this procedure, easily-retargetable high performance compilers play a key role in the design and development, improving productivity and reducing time-to-market. One of the more pressing challenges in this area is the need for easy-retargetability within the wide range of processors of the same processor family. This occurs due to the continual

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 121–136, 2011.

upgrading of instruction sets, as well as the fact that some processor families within the same architecture have incompatible instruction sets, such as occurs between the Power Family and PowerPC processors of the Power architecture.

The problem is further increased by the growing complexity of powerful and high-performance embedded processors. For instance, the high-performance embedded processor PowerPC, widely used in communication and aviation, has more than 200 instructions, and writing an architecture description language (ADL) for them requires handling significant complexity. This situation requires expressive, concise and high-quality architecture description languages for compilers. Unfortunately, most of the industrial-strength compilers lack easy-retargetability. Their retargeting work demands massive and error-prone modification of source code of the compilers which can take tremendous time, and also brings about expensive development costs.

Generating an auto-retargetable backend for a industrial-strength compiler is certainly difficult. There is auto-retargeting work for gcc [4], based on an ADL and tree-pattern matching [1], but it is limited to compiling medium-sized benchmarks, such as MiBench and MediaBench. There are also auto-retargetable compiler systems such as *CoSy* [17] and *Tensilica* [29], but we have not seen any published results of large-scale applications such as SPEC CPU on these compilers. Moreover, the architectures they worked on were simpler than PowerPC.

This paper proposes a novel, object-based ADL *OpenDL* and a well-structured *Rule Library* to automatically retarget compiler backends, ensuring efficient handling of complex instruction sets and rule patterns. With the introduction of object-based inheritance features, OpenDL not only is succinct and flexible, but also leads to a high-quality description that tends to have fewer bugs. With the concept of instruction template, the well-structured Rule Library allows detailed specification of instructions to handle complex rule patterns.

We use OpenDL and the Rule Library to automatically retarget the open source, industrial-strength compiler Open64 to the high performance embedded processor PowerPC. By introducing a well-defined intermediate representation interface *AutoAST* we generate a reliable version of an auto-retargeting, industrial-strength compiler for large-scale practical applications. It passes about the same number of gcc-torture cases as the newest gcc 4.5, and also achieves comparable performance to gcc 4.5 for both the embedded benchmark EEMBC and the general purpose benchmark SPEC CPU 2000. For one SPEC CPU benchmark program, it can generate 20% faster code than gcc.

2 Design Overview

2.1 Background

Open64. Open64 is a well-structured industrial-strength high performance open source compiler released under the GNU General Public License (GPL) targeting IA-32, x86-64 and IA-64 architectures. It originates with the SGI Pro64 open source compiler suite, evolving from SGI's product compiler MIPSPro. Recently, Open64 has been retargeted to MIPS [28], PowerPC [21], NVISA [25]

and Simplight [23] platforms, and has been used for development of an efficient DSP compiler [7].

Open64 has some powerful functional components to analyze and optimize programs [22]. Inter-procedural analysis (IPA) uses local dataflow information to construct the call graphs and perform alias analysis, dead code elimination, constant propagation, function inlining and global variable optimization. Loop-nest optimization (LNO) performs loop distribution, automatic vectorization and parallelization, *etc.*. Global optimization (WOPT) converts the main intermediate representation (WHIRL, an AST representation) to a hashed SSA form and performs optimizations such as partial redundancy elimination, pointer analysis, copy propagation and induction-variable recognition. Finally, the code generator (CG) performs instruction selection and scheduling, software pipelining, register allocation, architecture-dependent optimization and finally emits the assembly.

The different components of the compiler such as IPA, LNO, WOPT and CG communicate through the WHIRL AST. WHIRL has 5 levels of representation: Very High (VH), High (H), Mid (M), Low (L) and Very Low (VL). VH WHIRL contains all the semantic information in the source code; VL WHIRL only exists in code generators.

Olive. Code selectors of most auto-retargeting work use tree-pattern matching and dynamic programming techniques, first introduced in *twig* by Aho, et al. [12]. Fraser, et al. [9] put forward *burg,* applying bottom-up rewrite system (BURS) theory [27] to move the execution of dynamic programming to compile-compile time. This has functional limitations, and is difficult to understand and debug. Fraser, et al. [8] then proposed *iburg* which directly uses *switch* and *if* statements to perform pattern matching instead of applying BURS theory.

After comparing the merits of twig, burg and iburg, Steve Tjiang improved upon twig and created *Olive* [13]. Olive implements a similar powerful grammar to twig while performing pattern matching directly using *switch* and *if*, as is done in iburg. To the best of our knowledge, Olive is so far the most powerful code generator generator (CGG). As a result, we adopt Olive for automatic code generation in our auto-retargetable Open64 backend.

2.2 Design Goals

We have several design goals for the automatically retargetable compiler: usability, generality and performance.

To achieve superior usability, we use OpenDL to handle the complexity of writing and maintaining the architecture description. As OpenDL has object-based inheritance mechanisms, there is little redundancy in the description and most of the copy-paste work common in ADL editing is reduced. It is easy to automatically retarget the compiler to a new architecture using the Rule Library. By relying on instruction templates, it can handle the detailed instruction specification and complex rule patterns that typically confront automatic retargeting work necessary to direct industrial-strength compilers at powerful processors.

We introduce a well-defined intermediate representation interface *AutoAST* as a solution to our generality objective. It bridges the gap between the Open64 intermediate representation and the code generator generator (CGG) Olive [13]. AutoAST makes it easy to integrate Olive into the Open64 intermediate representation and into other compilers.

For the aim of high performance, we chose to automatically retarget the Open64 compiler, which is a powerful, industrial-strength high-performance compiler with many elaborate analysis and optimization components. Currently, it is the fastest compiler for SPEC CPU report on AMD processors. However, it was not designed to be easily retargetable, and retargeting work requires considerable effort for even similar architectures [6]. We chose to automatically retarget Open64 to utilize its strong high level optimizers as well as to facilitate its retargeting efforts.

2.3 Design Framework

The auto-retargetable code generator is illustrated in Figure 1. We maintain the Rule Library which contains instruction templates and grammar rule templates that will be discussed in section 4.4. OpenDL Analyzer reads the OpenDL description of the target architecture and then selects suitable rule templates from the Rule Library and generates corresponding grammar rules according to the OpenDL description, after which it outputs the grammar rules source file.

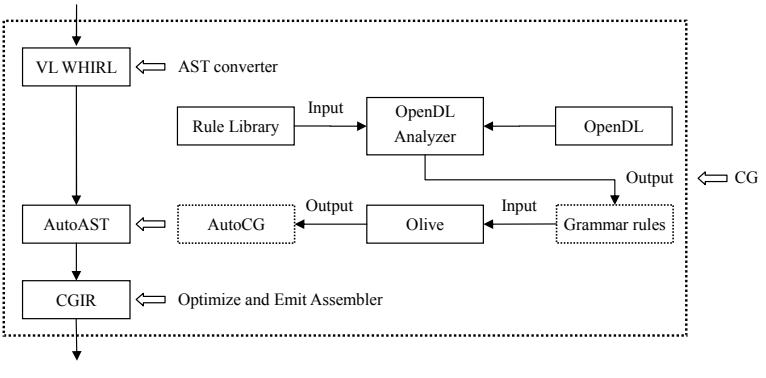


Fig. 1. Auto-Retargetable Code Generator

After the grammar rules are generated for the specified target, the code generator generator Olive compiles the grammar source file and generates source code for the code generator *AutoCG*. Finally AutoCG is compiled into the backend of the compiler.

When the auto-retargeted compiler compiles a source language, the AST converter transforms VL WHIRL to AutoAST at the code generation phase. Subsequently AutoCG applies tree-pattern matching and dynamic programming on AutoAST to perform optimized code generation.

3 OpenDL

OpenDL is an object-based ADL with XML-like grammar. By introducing the inheritance mechanisms of Common Information Extraction, Instruction Sharing and Pattern Abstraction, it leads to a concise and high-quality description of instructions.

3.1 Instruction Description

Figure 2(a) demonstrates the definition of an instruction. In the operand section, the operands of the instructions are listed in order, and the type field specifies the operand machine type. The action section describes the semantics of the instruction. In this case it describes a kind of addition instruction, with the semantics that the destination operand is assigned the sum of the two source operands. The assembly part describes how the instruction is written in the assembly language: "%name" represents the name of the instruction ("addi" in this case), and each "%s" is a given operand in order. The encoding section specifies how the instruction is binary encoded for the processor. The specification is in a concise segmented form. Each segment is comprised of two parts, where the first represents the content of the segment and which can be a fixed number or the encoding of an operand, and the second is the occupied bits of the segment, presented in brackets.

3.2 Common Information Extraction

Though there are a number of instructions in modern embedded processors, many of them are similar and can be sorted into only a few classes. For example, many architectures supply integer and floating-point arithmetic instructions, bit-operation instructions, memory access instructions, control instructions such as data move into and out of control registers, SIMD instructions, etc. For example, bit-operations such as *and*, *or*, *nand* (not and), *nor* (not or) and *xor* have both logical and practical similarities. The common information can be extracted into separate classes and instruction descriptions can then reuse the same classes. In this way, the description can be much more succinct and the design is more flexible and easier to modify.

The common information extraction inheritance mechanism is illustrated in Figure 2(b), where the class keyword defines the class inherited by these instructions, here "BitOperation," and the base attribute denotes the super-class, in this case the built-in class "InstructionSet."

3.3 Instruction Sharing

There are also instructions which act like each other, but with some variation in routine behaviors. An example is the PowerPC instruction syntax form. Most instructions have two or four syntax forms which affect flag registers. Instructions such as *add*, *addc* and *mul* have four syntax forms, i.e. *add*, *add.*, *addo*, *addo.*,

```
<InstructionSet>
  <addi>
    <operand>
      <result type="GPR" />
      <src1 type="GPR" />
      <src2 type="simm16" />
    </operand>
    <action> result = ADD(src1, src2) </action>
    <assembly pattern="%name %s, %s, %s" />
    <encoding>
      14 [0-5] result [6-10] src1 [11-15] src2 [16-31]
    </encoding>
    …
  </addi>
  …
</InstructionSet>
```

(a) Instruction Description

```
<class name="BitOperation"
       base="InstructionSet"
       operator="op">
  <action> result = op(src1, src2) </action>
  <assembly pattern="%name %s, %s, %s" />
  …
</class>
<and inherit="BitOperation" operator="AND" />
<or inherit="BitOperation" operator="OR" />
```

(b) Common Information Extraction

```
<class name="Syntax4",
       base="InstructionGroupGenerator">
  <form suffix="" />
  <form suffix =".">
    <action>
      $action(%instruction)
      testset CR0[LT, GT, EQ, SO]
    </action>
    <encoding> 1 [31] </encoding>
  </form>
  …
</class>
<add syntax="Syntax4">
  …
</add>
```

(c) Instruction Sharing

```
<class name="ImmediateDecorator"
       base="Decorator" suffix="i">
  <operand>
    <src2 type="simm16" />
  </operand>
</class>
<class name="ExtendedDecorator"
       base="Decorator" suffix="e">
  <action>
    result = ADD($action(%instruction), CARRY)
    testset CARRY
  </action>
</class>
<add>
  <decorator name="" syntax="Syntax4" />
  <decorator name="ExtendedDecorator"
             syntax="Syntax4" />
  <decorator name="ImmediateDecorator" />
  …
</add>
```

(d) Pattern Abstraction

Fig. 2. OpenDL Description

etc.. The "*." form affects the LT, GT, EQ, SO of the CR0 register, "*o" form affects SO, OV of the fixed-point exception register (XER), "*o." form affects all the above, and "*" form affects none.

In order to describe the group of instructions, traditionally we need to define all the four instructions, which is redundant and painstaking. In OpenDL we can define the syntax form class which is an instruction group generator that automatically generates the four instructions; afterwards we need just define the primitive instruction once and designate the syntax form to the instruction. This shrinks the description, and has less redundancy.

Figure 2(c) illustrates instruction sharing. The syntax attribute indicates the syntax form class, and "$action(%instruction)" tells the generator to perform the instruction's own action first.

3.4 Pattern Abstraction

For most architectures, several instructions may perform the same type of operation, forming a decorator design pattern. One instruction may have several decorators and one pattern may apply to many types of instructions. Take the instructions performing add operations in PowerPC for example: *add*, *addc* (add carrying), *adde* (add extended), *addi* (add immediate), *addic* (add immediate carrying), *addis* (add immediate shifted). The carrying decorator means setting the CARRY bit, the extended decorator using and setting CARRY bit, and the shifted decorator logically shifting left the immediate value by 16 bits. The carrying and extended decorators also apply to *subf* instructions and the immediate and shift decorators are employed in *mul*, *and*, *or*, etc. We can abstract the decorators as semantic classes, and then all we need to do is define one instruction and give it the decorators to describe these instructions.

Figure 2(d) illustrates the combination of instruction sharing and pattern abstraction. Decorators are defined in their own classes and can be applied to instructions, each with independent syntax forms.

4 Rule Library

The Rule Library contains two parts: abstract machine instructions which are defined as instruction templates, and grammar rule templates which specify how the Open64 VL WHIRL AST maps to lists of abstract machine instructions. This section introduces the definition of rule templates and then presents the techniques used in rule mappings.

4.1 Terminal Declaration

Each WHIRL operation is specified by an opcode which consists of three components: operator, result type and descriptor type. Operator specifies the kind of the operation, while result type and descriptor type qualify the data type of the result and operands of the operation respectively.

We define terminals according to the opcode of the WHIRL node. Take the 32-bit integer add as an example, the opcode is OPC_I4ADD, and we define a terminal I4ADD for it. We generally define different terminals for different opcodes, but for opcodes of some operators such as load/store operations, we are not concerned with the descriptor type of the opcode, and thus define the same terminal for them to make the rules simpler. For example, we define the same terminal I4LDBITS for OPC_I4I1LDBITS, OPC_I4I2LDBITS, etc.

4.2 Nonterminal Declaration

We define nonterminals according to the result type of the semantic actions of the production. As the target is a RISC processor whose memory access operations are implemented by load and store instructions, the operands of other

instructions are all registers and/or immediates. We invoke external functions for load and store operations which simplifies the definition for nonterminals. Except for the start symbol, chief nonterminals are sorted into two categories of registers and immediates. Since 8-bit and 16-bit registers in CISC processors do not exist in our target, the register category consists of *reg*, *reg64*, *f4reg* and *f8reg*, while *imm16* and *imm32* comprise the immediate category.

4.3 Grammar Rules

A grammar rule contains two parts: a cost specification and an action specification. Currently, we use the sum of the clock cycles needed by the executed instructions of the semantic action as the cost. Other indicators can also be used for the cost function, such as the generated code size of the semantic action. For different scenarios, weighted sum of these indicators is still an alternative.

One production rule is illustrated in Figure 3. I4INTCONST, U4INTCONST, etc. are terminals discussed in section 4.1. The production claims that imm16 can be reduced by any of these terminals.

```
imm16 : I4INTCONST, U4INTCONST, I8INTCONST, U8INTCONST
        {
            CHECK(Has_Immediate_Operand($1->parent, $1->wn));
            CHECK(IN_RANGE(WN_const_val($0->wn), SIMM16));
            $cost[0].cost = 0;
        }
        = {
            $0->result = Gen_Literal_TN(WN_const_val($1->wn), 4);
        };
```

Fig. 3. A Grammar Rule

The cost is specified in the first brace. The dynamic programming algorithm calculates the minimum-cost match by the cost statements. The function CHECK checks if the production is valid, and if not, ignores the production. As shown in this production, Has_Immediate_Operand is an Open64 defined function which tests whether the parent WHIRL node can accept an immediate operand. "CHECK(IN_RANGE(WN_const_val($0->wn), SIMM16))" checks if the constant operand in the node is a 16-bit integer. After passing these two tests, it is assured that using this production for reduction to *imm16* is safe. The semantic action is the statement in the second brace "$0->result = Gen_Literal_TN(WN_const_val ($1->wn), 4);", which directly builds an immediate TN node with no instruction generation needed. The "$cost[0].cost=0;" declaration tells Olive the cost is 0 and using this production is encouraged.

4.4 Mapping Rules

Figure 4 illustrates grammar rules mapping. The production shows that *reg* can be reduced to a 32-bit integer subtraction, in which the minuend is *imm16* and the subtrahend is *reg64t*.

To map the grammar rules, we introduce instruction templates which are defined before the rule templates. An instruction template consists of a placeholder for an instruction and a specification of the characteristics of the instruction. In figure 4, the "subfic" after the "<%" is the placeholder name, and the instruction template can represent any instruction that has compatible operands and actions with those specified by the template.

The concept of instruction template in our rule mapping approach splits instruction specification and analysis from the rule templates. This is a well-structured approach that allows more detailed specification of instruction templates without introducing much complexity.

After instruction templates are defined, the grammar rule templates are provided. Each rule template as an annotation to specify the mapping, which is the placeholder name of an instruction template. The annotation indicates that the rule template refers to the corresponding instruction template. The semantic action thus can use the specified instruction.

One-to-many mapping, i.e. multiple IR operations mapping to one single instructions, can already be handled by the Olive grammar. If a rule needs more than one instructions, then all the instruction templates are annotated. In this way we can handle various cases, including many-to-one or even many-to-many mappings.

If the OpenDL Analyzer finds that a target processor supplies an instruction compatible with the instruction template then the rule template is selected and the instruction placeholder is replaced with the real instruction.

```
<% subfic
      <operand>
            <result type="GPR" />
            <src1 type="GPR" />
            <src2 type="imm16" />
      </operand>
      <action> result = SUB(src2, src1) testset CARRY </action>
%>
…
reg   : I4SUB(imm16, reg64t), U4SUB(imm16, reg64t)
      {
            $cost[0].cost = 1 + $cost[3].cost;
      }
      = { <% subfic %>
            $action[2](ops);
            $action[3](ops);
            Build_OP(subfic, $0->result, $3->result, $2->result, ops);
      };
```

Fig. 4. Rule Mapping

5 AutoAST

AutoAST is a well-defined intermediate representation interface isomorphic to the WHIRL tree, which is comprised of an AST converted from Open64 VL

WHIRL and attributes and methods utilized by AutoCG. AutoCG executes tree-pattern matching and uses a dynamic programming algorithm on AutoAST to perform automatic code generation.

Olive performs the tree-pattern matching according to an attribute *terminal* of each node of the AST. If an AST satisfies the requirement of having the following 4 methods, it can be used by Olive for automatic semantics generation: (1) set_state(burm_state* s), set state s to the node. (2) burm_state* state_label(), return the state of the node. (3) NODEPTR get_kids(), return the pointer array containing the kids of the node. (4) int op_label(), return the *terminal* of the node. Other needed data fields and methods may be added to implement the function of code generation.

As the CGG Olive requires and maintains the attributes *kids*, *state* and *label*, they are stored and made accessible to Olive through its interface. The intermediate variable *result* computed by semantic action of the corresponding sub-tree should also be stored. To access the WHIRL data, the corresponding WHIRL node *wn* is saved. The parent WHIRL node is also saved as it is required by the code generator. Commonly used data such as the opcode and operator of the WHIRL node are also saved in the node.

Before being transformed to CGIR, VL WHIRL is converted to AutoAST by the AST converter. Each node is allocated and visited in a preorder traversal, where we also assign its attributes to their corresponding values. Once the traversal is completed, the AutoAST is constructed.

6 Experimental Results

We have implemented two Open64 backends on the PowerPC architecture. One is the auto-retargeted version and the other is a manually retargeted one. So far both these implementations have achieved sound correctness and fine performance. Our effort is approximately 4 person-years of work and over 0.2 million lines of code.

The experiments are performed on an Apple iBook G4 computer with PowerPC 7447A processor of a frequency of 1.33GHz and 1 GB memory. The operating system is Ubuntu 8.04, with kernel 2.6.24-23-powerpc and gcc 4.5.

6.1 OpenDL

In the common instruction set of the PowerPC architecture there are 24 instructions with 4 syntax forms, 44 instructions with 2 syntax forms and 109 instructions with 1 syntax form. Using a traditional description, we need to write $24 * 4 + 44 * 2 + 109 = 293$ instructions. However, merely by using the simplest inheritance mechanism of instruction sharing the number sharply decreases to $24 + 44 + 109 = 177$, a compression rate of $177/293 = 60.4\%$. With a combination of instruction sharing and pattern abstraction, the required instruction number can be further reduced to 100, resulting in a compression rate up to $100/293 = 34.1\%$. Finally, if the common information extraction is also utilized, the rate is reduced to below 25%. The result is illustrated in Figure 5.

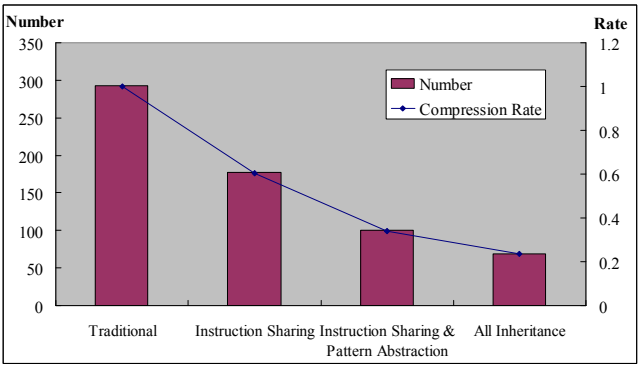


Fig. 5. OpenDL Compression Results

6.2 Reliability

We first test the correctness of the retargeted backends. The test suite we select is gcc-torture 3.3.6 which is used by gcc as an important evaluation of reliability. The results are shown in Table 1. The number of test cases we have passed is comparable to that of gcc. Although for some cases we fail due to non-standard gcc extensions, we feel this is substantial evidence that the retargeted compilers are reliable for a wide range of situations.

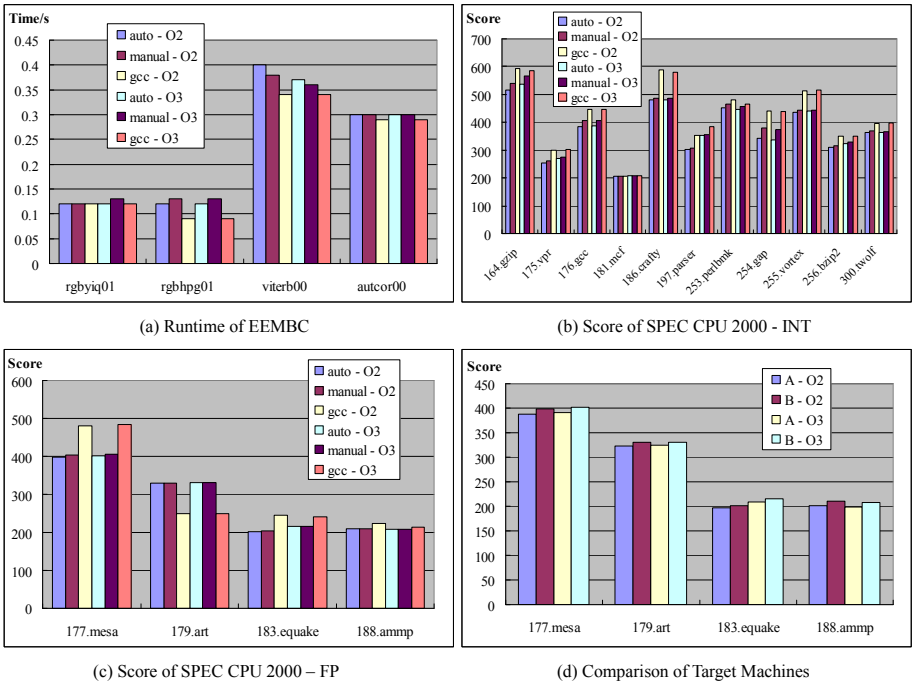
Table 1. Gcc-torture Results

Good cases	backend	Build passed	Build failed	Run cases	Run passed	Run failed
O0 1525	Auto	1428	97	741	677	64
	Manual	1428	97	741	677	64
	Gcc	1480	45	769	710	59
O2 1525	Auto	1434	91	746	663	83
	Manual	1434	91	746	663	83
	Gcc	1465	60	753	687	66
O3 1525	Auto	1433	92	744	675	69
	Manual	1433	92	744	675	69
	Gcc	1465	60	753	687	66

6.3 Performance of Embedded Benchmarks

For performance we experiment on the well-known embedded EEMBC benchmark suite. The running time of some of the Consumer and Telecom benchmarks are shown in Figure 6(a). The auto bars represent the results of the automatic retargeted backend, the manual bars present the results of the manually implemented version, and the gcc bars show the gcc results.

For O2 and O3, most of the programs achieve similar performance to the gcc versions. It also shows that the automatic retargeted backend generates



(a) Runtime of EEMBC

(b) Score of SPEC CPU 2000 - INT

(c) Score of SPEC CPU 2000 – FP

(d) Comparison of Target Machines

Fig. 6. Performance Results

faster code than the manual implementation for many programs. For rgbhpg01, the auto-retarget backend beats the manual implementation for all optimization levels and we think instruction selection optimizations are more essential to it.

6.4 Performance of Large-Scale Applications

We have also measured performance on large-scale applications, using SPEC CPU 2000. Figures 6 (b) and (c) present a comparison of the benchmark scores of the two retargeted implementations of the Open64 compiler and gcc.

It can be seen that the performance of the auto-retarget implementation is close to the manual implementation, with a difference of less than 5%. We note that the manually retargeted backend has undergone substantial optimization, and thus we consider the performance of the auto-retargeting to be satisfactory.

From the results shown in the figures, for large applications, the retargeted compilers also achieve performance close to gcc for most programs. Furthermore, for the FP program 179.art, both implementations produce much faster code than gcc.

6.5 Case Study: Design Space Exploration of ASIPs

As a final case study, we experiment with the common scenario of design space exploration in ASIPs. For ASIP design, often a basic processor and the corresponding compiler is readily available, yet tailoring the processor and compiler is required for specific applications. In this process there are usually many rounds of adding and removing some accelerating instructions and experimenting with the results.

We suppose to have an ASIP processor A with PowerPC instruction set yet without the compound arithmetic instructions. We want to accelerate special floating point applications, and therefore we attempt to produce a processor B that adds some of the compound floating point arithmetic instructions of *fmadd*, *fmsub*, *fnmadd*, *fnmsub*, *fmadds*, *fmsubs*, *fnmadds*, *fnmsubs*, and perform some experiments on the processors using the tailored compilers.

The rule maintenance work is simple. If rule templates for these instructions are in the Rule Library, no additional work is needed. However, if these instructions are comparatively new and are absent from the Rule Library, to utilize the special instructions we supply additional rule templates for these compound arithmetic instructions. The work is just addition of one rule template for each tree pattern corresponding to the compound arithmetic instructions; that is, a total of 8 grammar rules and some dozens of lines-of-code.

The retarget time is merely the time for modification of the OpenDL description, which is negligible, and the time needed to recompile the code generator and link the compiler—several minutes on our experimental machine. In contrast, for a traditional compiler we need large-scale and error-prone modification to the compiler's source code, as well as extensive regression testing. In our case, since no rule templates contain more than one of these instructions, we can easily retarget the compiler to processors with any combination of these instructions—a total of $2^8 = 256$ targets—with just small changes to the OpenDL description of the target architecture, a trivial process that requires no modifications to the Rule Library.

We then run the SPEC2000 benchmark on the two processors and compare the results. Figure 6(d) shows the scores given by the SPEC FP programs on the basic processor A and a new processor B equipped with the 8 compound floating point arithmetic instructions. We can see that the new processor runs faster than the basic processor for these applications, as expected.

The auto-retargetable compiler can produce correct code for target processors with the guarantee that instructions not specified in the ADL are never emitted, without potentially introducing bugs to other parts of the compiler. Processor designers can securely remove instructions from processors, and are assured these processors will behave correctly with no performance reduction for applications without floating point operations. This has the additional benefit that a compact instruction set makes the processor more energy efficient and may even get faster by being able to otherwise exploit the space saved by removing the redundant instructions.

7 Related Work

There have been many existing attempts to provide compiler descriptions and facilities to aid in retargeting. MIMOLA [3], for instance, is an HDL-like architecture description language used in the MSSQ [20] compiler. Brander, et al. [4] put forward another structural ADL following a component based paradigm to retarget the gcc compiler with iburg. nML [10] and ISDL [11] use an attribute grammar to describe the instruction set. CHESS [19] used nML to describe fixed-point DSPs and ASIPs. ISDL was primarily designed for VLIW architectures and has been used in the AVIV [16] compiler's retargetable code generator.

MDes [14] and EXPRESSION [15] have been primarily targeted at design space exploration. MDes has been used to describe the HPL-PD machine in Trimaran compilation and performance monitoring infrastructure [30]. EXPRESSION was employed for software toolkit generation for architectural exploration of programmable SOCs [24]. ArchC [2] is an ADL focused on SystemC users. Simulators and assemblers of some architectures have been generated using ArchC.

LISA [26] was designed primarily for compiled cycle- and bit-accurate fast simulators. However, LISA lacks the high-level semantics required by compiler retargeting since it is generally unable to extract this information from the arbitrary C statements. Ceng, et al. [5] extended the LISA description by adding instruction semantics for the compiler which was used to retarget the CoSy compiler's code selector [17]. An instruction scheduler was extracted from cycle-true LISA processor models in [31]. Hohenauer, et al. [18] retargeted the CoSy compiler with a GUI-based semi-automatic approach using LISA description.

Lin, et al [21] retargeted Open64 to PowerPC with Olive, passing half of the SPEC2000 CINT benchmarks, and this provided a basis for our work. We use OpenDL to automatically retarget the compiler and generate the first reliable version of Open64 on PowerPC architecture.

8 Conclusion

In this paper we proposed OpenDL, an object-based ADL, along with a well-structured Rule Library to automatically retarget compilers, efficiently handling complex architecture description and rule patterns. By applying novel, object-based inheritance features, OpenDL not only is succinct and flexible, but also leads to a high-quality description that tends to have fewer bugs. With the concept of an instruction template, the well-structured Rule Library allows detailed, rule-based specification of instructions.

As a proof of concept, we automatically retargeted the open source industrial-strength compiler Open64 to the high performance embedded processor PowerPC using OpenDL and the Rule Library, and compared results with manual retargeting. Experimental results show that both manually and automatically retargeted compilers are sound, reliable and efficient, passing about the same number of gcc-torture cases as the newest gcc 4.5. The performance of the code generated by the compilers are also comparable to gcc 4.5 for most applications.

For one SPEC CPU benchmark program, the compilers can generate code that is 20% faster than gcc.

In future work, we will strengthen OpenDL with more features. We will also automatically retarget the Open64 compiler to other RISC processors or other architectures such as DSP and VLIW processors. We may also use different cost calculations, combining performance and code size to make multi-objective optimizations.

Acknowledgments. We thank Professor Clark Verbrugge of McGill University and the anonymous reviewers for their comments to improve the paper. We also thank Duo Zhang and Zhenyang Yu for their related work. This work is supported in part by National Natural Science Foundation of China (No. 90818019, No. 61170051) and Hi-Tech Research and Development Program of China (No. 2008AA01Z102). Any opinions, findings, and contributions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Aho, A.V., Ganapathi, M., Tjiang, S.W.K.: Code generation using tree matching and dynamic programming. ACM Transactions on Programming Languages and Systems 11(4), 491–516 (1989)
2. Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C., Barros, E.: The archc architecture description language and tools. International Journal of Parallel Programming 33(5), 453–484 (2005)
3. Bashford, S., Bieker, U., Harking, B., Leupers, R., Marwedel, P., Neumann, A., Voggenauer, D.: The mimola language, version 4.1. University of Dortmund (September 1994)
4. Brandner, F., Ebner, D., Krall, A.: Compiler generation from structural architecture descriptions. In: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 13–22 (October 2007)
5. Ceng, J., Sheng, W., Hohenauer, M., Leupers, R., Ascheid, G., Meyr, H.: Modeling instruction semantics in ADL processor descriptions for C compiler retargeting. Journal of VLSI Signal Processing 43, 235–246 (2006)
6. Cui, H., Feng, X.: Retargeting open64 to a RISC processor – a students perspective. In: Open64 Workshop at CGO (2008)
7. De, S.K., Dasgupta, A., Kushwaha, S., Linthicum, T., Brownhill, S., Larin, S., Simpson, T.: Development of an efficient DSP compiler based on open64. In: Open64 Workshop at CGO (2008)
8. Fraser, C.W., Hanson, D., Proebsting, T.A.: Engineering a simple, efficient code generator generator. ACM Letters on Programming Languages and Systems 1(3), 213–226 (1992)
9. Fraser, C.W., Henry, R.R., Proebsting, T.A.: Burg: Fast optimal instruction selection and tree parsing. ACM SIGPLAN Notices 27(4), 68–76 (1992)
10. Freericks, M.: The nml machine description formalism, version 1.5. TU Berlin Computer Science Technical Report (1993)
11. George Hadjiyiannis, S.H., Devadas, S.: Isdl: An instruction set description language for retargetability. In: Design Automation Conference, pp. 299–302 (1997)

12. Graham, S.L.: Table-driven code generation. Computer 13(8), 25–34 (1980)
13. Group, S.R.: Spam compiler user's manual (1997)
14. Gyllenhaal, J.C.: A machine description language for compilation. Master Thesis, Department of EE, UIUC (1994)
15. Halambi, A., Grun, P., Ganesh, V., Khare, A.: Expression: A language for architecture exploration through compiler/simulator retargetability. In: Design, Automation and Test in Europe, pp. 485–490 (1999)
16. Hanono, S., Devadas, S.: Instruction selection, resource allocation, and scheduling in the aviv retargetable code generator. In: Design Automation Conference, pp. 510–515 (1998)
17. Hohenauer, M., Engel, F., Leupers, R., Ascheid, G., Meyr, H., Bette, G., Singh, B.: Retargetable code optimization for predicated execution. In: Design, Automation and Test in Europe, pp. 1492–1497 (2008)
18. Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G., van Someren, H.: A methodology and tool suite for C compiler generation from ADL processor models. In: Design, Automation and Test in Europe, vol. 2, pp. 21–26 (2004)
19. Lanneer, D., Praet, J.V., Kifli, A., Schoofs, K., Geurts, W., Thoen, F., Goossens, G.: Chess: Retargetable code generation for embedded DSP processors. Code Generation for Embedded Processors, pp. 85–102 (1995)
20. Leupers, R., Marwedel, P.: Retargetable code generation based on structural processor descriptions. Design Automation for Embedded Systems 3(1), 75–108 (1998)
21. Lin, M., Yu, Z., Zhang, D., Zhu, Y., Wang, S., Dong, Y.: Retargeting the open64 compiler to powerpc processor. In: The 2008 International Conference on Embedded Software and Systems Symposia, pp. 152–157 (2008)
22. Liu, S.-M.: A tutorial: Open64 release 4.0: High performance compiler for itanium and x86 linux. In: PLDI (2007)
23. Lo, K.M., Ma, L.: Quantitative approach to ISA design and compilation for code size reduction. In: Open64 Workshop at CGO (2008)
24. Mishra, P., Shrivastava, A., Dutt, N.: Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable socs. ACM Transactions on Design Automation of Electronic Systems 11(3), 626–658 (2006)
25. Murphy, M.: Nvidia's experience with open64. In: Open64 Workshop at CGO (2008)
26. Pees, S., Hoffmann, A., Zivojnovic, V., Meyr, H.: Lisa - machine description language for cycle-accurate models of programmable DSP architectures. In: Design Automation Conference, pp. 933–938 (1999)
27. Proebsting, T.A.: Simple and efficient burs table generation. In: PLDI, pp. 331–340 (1992)
28. Shuchang, Z., Ying, L., Fang, L., Le, Y., Lei, H., Shuai, L., Chunhui, M., Zhitao, G., Ruiqi, L.: Open64 on mips: porting and enhancing open64 for loongson ii. In: Open64 Workshop at CGO (2008)
29. Tensilica, http://www.tensilica.com
30. Trimaran: A compiler and simulator for research on embedded and epic architecture, version 4.0 (April 2007), http://www.trimaran.org/docs/trimaran4_manual.pdf
31. Wahlen, O., Hohenauer, M., Braun, G., Leupers, R., Ascheid, G., Meyr, H., Nie, X.: Extraction of Efficient Instruction Schedulers from Cycle-True Processor Models. In: Anshelevich, E. (ed.) SCOPES 2003. LNCS, vol. 2826, pp. 167–181. Springer, Heidelberg (2003)

A Non-iterative Data-Flow Algorithm for Computing Liveness Sets in Strict SSA Programs

Benoit Boissinot[1], Florian Brandner[1], Alain Darte[1], Benoît Dupont de Dinechin[2], and Fabrice Rastello[1]

[1] LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
[2] Kalray

Abstract. We revisit the problem of computing liveness sets (the sets of variables live-in and live-out of basic blocks) for programs in strict static single assignment (SSA). In strict SSA, aka SSA with dominance property, the definition of a variable always dominates all its uses. We exploit this property and the concept of loop-nesting forest to design a fast two-phases data-flow algorithm: a first pass traverses the control-flow graph (CFG), propagating liveness information backwards, a second pass traverses a loop-nesting forest, updating liveness sets within loops. The algorithm is proved correct even for irreducible CFGs. We analyze its algorithmic complexity and evaluate its efficiency on SPECINT 2000. Compared to traditional iterative data-flow approaches, which perform updates until a fixed point is reached, our algorithm is 2 times faster on average. Other approaches are possible that propagate from uses to definitions, one variable at a time, instead of unioning sets as in data-flow analysis. Our algorithm is 1.43 times faster than the fastest alternative on average, when sets are represented as bitsets and for optimized programs, which have non-trivial live-ranges and a larger number of variables.

1 Introduction

Static single assignment (SSA) is a popular program representation used by most modern compilers. Initially developed to facilitate the development of high-level program transformations, SSA has gained much interest due to its properties that often lead to simpler algorithms and reduced computational complexity. Today, SSA is even adopted for the final code generation phase [26]. For instance, several industrial and academic compilers, static or just-in-time, use SSA in their back-ends, e.g., LLVM [28], Java HotSpot [25], LAO [17], LibFirm [27,14], Mono [31]. Recent research on register allocation [10,18,33] showed that SSA form can even be retained until the very end of the code generation process.

This work explores the use of SSA properties to simplify and accelerate *liveness analysis*, which determines, for each basic block, the variables whose values are eventually used by subsequent operations. This information is essential to solve storage assignment problems, eliminate redundancies, and perform

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 137–154, 2011.

code motion. Optimizations like software pipelining, trace scheduling, register-sensitive redundancy elimination, if-conversion, and register allocation heavily rely on liveness information. Traditionally, liveness is obtained by data-flow analysis: liveness sets are computed for all basic blocks, all variables treated together, by solving a set of data-flow equations [3]. These equations are usually solved by an iterative algorithm, propagating information backwards through the control-flow graph (CFG) until a fixed point is reached. The number of iterations depends on the CFG structure and on the order in which basic blocks are evaluated.

We show that, for SSA-form programs, it is possible to design a fairly simple, two-passes, data-flow algorithm to compute liveness sets *that does not require to iterate* to reach a fixed point. Its first pass, very similar to the initialization phase of traditional data-flow analysis, computes partial liveness sets by traversing the CFG backwards. Its second pass refines the partial liveness sets within loops by traversing a loop-nesting forest, as defined by Ramalingam [35]. Such a loop hierarchy is already available in modern compilers for other optimizations that exploit the structure of loops. For the sake of clarity, we first present our algorithm for reducible CFGs, then we show that irreducible CFGs can be handled with a slight variation, with no need to modify the CFG itself.

Other approaches are possible, for example as proposed by Appel [3] or McAllester [29], that propagate liveness from uses to definitions, one variable at a time, instead of unioning sets as in standard data-flow analysis. For a broader comparison with state of the art, we designed optimized implementations of this path-exploration principle (improved to work at the granularity of basic blocks instead of instructions as in the original versions) and compared the efficiency of the resulting algorithms with our non-iterative data-flow algorithm.

Our experiments using the SPECINT 2000 benchmark suite demonstrate that the non-iterative data-flow algorithm outperforms the standard iterative data-flow algorithm by a factor of 2 on average. By construction, our algorithm is best suited for a set representation, such as bitsets, favoring operations on whole sets. In particular, for optimized programs, which have non-trivial live-ranges and a larger number of variables, our algorithm achieves a speed-up of 1.43 on average in comparison to the fastest alternative based on path exploration.

2 Related Work

Liveness information is usually computed with iterative data-flow analysis, which goes back to Kildall [24]. The algorithms are, however, not specialized to the computation of liveness sets and may incur overhead. Several strategies are possible, leading to different worst-case complexities and performance in practice. *Round-robin algorithms* propagate information according to a fixed block ordering derived from a depth-first spanning tree and iterate until it stabilizes. The complexity of this scheme was analyzed by Kam et al. [23], see Section 3.4. *Worklist algorithms* focus on blocks that may need to be updated because the liveness sets of their successors (for backward problems) changed. Empirical results by Cooper et al. [15] indicate that the order in which basic blocks are processed

is critical and directly impacts the number of iterations. They showed that, in practice, a mixed solution, called "single stack worklist", based on a worklist initialized with a round-robin order, is the most efficient one for liveness analysis. In contrast, our non-iterative data-flow algorithm requires at most two passes over the basic blocks, in all cases. In practice, for strict SSA programs, it is on average twice as fast as the "single stack worklist" approach (see Section 5).

An alternative way to solve data-flow problems is interval analysis [2] and other elimination-based approaches [36]. The initial work on interval analysis [2] demonstrates how to compute liveness information using only three passes over the *intervals* of the CFG. However, the problem statement involves, besides the computation of liveness sets, several intermediate problems, including separate sets for reaching definitions and upward-exposed uses. Furthermore, the number of intervals of a CFG grows with the number of loops. Also, except for the Graham-Wegman algorithm, interval-based algorithms require the CFG (resp. the reverse CFG) to be reducible for a forward (resp. backward) analysis [36]. In practice, irreducible CFGs are rare, but liveness analysis is a backward data-flow problem, which frequently leads to irreducible reverse CFGs. In contrast, our algorithm does not require the *reverse* CFG to be reducible. If the CFG is irreducible, care must be taken when propagating liveness information backward in the CFG, but with no modification of the CFG itself (see Section 4.2).

Another approach to compute liveness was proposed by Appel [3, p. 429]. Instead of computing the liveness information for all variables at the same time, variables are handled individually by exploring paths in the CFG starting from variable uses. An equivalent approach using logic programming was presented by McAllester [29], showing that liveness analysis can be performed in time proportional to the number of instructions and variables. However, his theoretical analysis is limited to a restricted input language with simple conditional branches and instructions. A more generalized analysis will be given later, both in terms of theoretical complexity (Section 3.4) and of practical evaluation (Section 5).

Liveness analysis for strict SSA programs was first addressed by Boissinot et al. [9], but with a different perspective. They showed that queries such as "is variable v live at program point p?" can be performed quickly, thanks to a pre-processing step depending on the CFG structure only. Wimmer et al. [39] gave an algorithm, specialized to linear scan register allocation, to build the "intervals" of basic blocks where each variable is live. Although a possible extension to irreducible CFGs is sketched, the algorithm restricts itself to reducible CFGs or to a form of SSA where live-ranges are cut at loop-entry blocks with ϕ-functions. The algorithm we propose is a generalization[1] for computing liveness sets: it uses the concept of loop-nesting forest and is proved correct with no restriction on the CFG, on the strict SSA form, or on the loop-nesting forest as long as it respects the minimal properties stated by Ramalingam [35]. As a by-product, this proves the correctness of the algorithm of [39] and how a suitable order of basic blocks can be chosen thanks to a loop-nesting forest. Such orders were

[1] Actually, we designed this algorithm in 2009-2010 independently of [39].

also exploited for liveness analysis in static single information (SSI) [8]. The live-check algorithm of [9] was also reformulated using loop-nesting forests [6].

3 Foundations

This section recalls the concepts of control-flow graphs, loop-nesting forests, dominance, and SSA form. Readers familiar with them can skip this section.

3.1 Control-Flow Graph and Loop Structure

A *control-flow graph* $G = (V, E, r)$ is a directed graph, with nodes V, edges E, and a distinguished *root* $r \in V$ from which there is a path to any other node. Usually, the CFG nodes represent the basic blocks of a procedure or function, every block is in turn associated with a list of operations or instructions.

Dominance. A node x in a CFG *dominates* a node y if every path from the root r to y contains x. The dominance is strict if $x \neq y$. The transitive reduction of the dominance relation forms a tree, the *dominator tree*.

Loop-Nesting Forest. To discuss previously-proposed constructions, Ramalingam [35] gave a minimal definition of loop-nesting forest by a recursive process:

1. Partition the CFG into its strongly connected components (SCCs). Every non-trivial SCC, i.e., with at least one edge, is called a *loop*.
2. For each loop L, select a non-empty set of nodes in L among those that are not dominated by any other node in L: its elements are called the *loop-headers* of L. (Different choices may lead to different forests.) Remove all edges in L that lead to a loop-header of L and call them the *loop-edges* of L.
3. Repeat this partitioning recursively for every SCC, after its loop-edges have been removed. The process stops when only trivial SCCs remain.

This decomposition can be represented by a forest whose leaves are the nodes of the CFG, while internal nodes, labeled by loop-headers, correspond to loops. The children of a loop's node represent all inner loops (i.e., all non-trivial SCCs) it contains as well as the regular basic blocks of the loop's body. The forest can easily be turned into a tree by introducing an artificial root node, corresponding to the entire CFG. Note also that a loop-header cannot belong to any inner loop because all edges leading to it are removed before computing inner loops. In the rest of this paper, we make no other assumption on the way the loop-nesting forest is built: any of the algorithms analyzed in [34,35] can be applied.

Reducible Control-Flow Graphs. A CFG is *reducible* if every loop has a single node that dominates all other nodes of the loop [20]. In other words, the only way to enter a loop is through its unique loop-header. Because of its structural properties, the class of reducible CFGs is of special interest for compiler writers. Indeed, most programs exhibit reducible CFGs. Also, as pointed out

earlier, unlike other approaches that compute liveness information, we only need to discuss the reducibility of the original CFG, not of the reverse CFG.

Computing a Loop-Nesting Forest. A loop-nesting forest can be computed in "almost linear time" $O(|E|\alpha(|E|, |V|))$ [34]. Unlike reducible CFGs, the loop-nesting forest of an irreducible CFG is not unique as some loops have several nodes not dominated by any other node in the loop. A simple-to-engineer construction algorithm is the generalization of Tarjan's algorithm [38] proposed by Havlak [19], later improved by Ramalingam [34] to fix a complexity issue. It identifies a loop as a set of descendants of a back-edge target that can reach its source. In that case, the set of loop-headers is restricted to a single entry node, the target of a back-edge. Also, while identifying loops, whenever an entry node that is not the loop-header is encountered, the corresponding incoming edge (from a non-descendant node) is replaced by an edge to the loop-header.

3.2 Static Single Assignment Form

Static single assignment (SSA) [16] is a popular program representation. In SSA, each scalar variable is defined only once textually. To build SSA, variables having multiple definitions are replaced by several new *SSA variables*, one for each definition. When a use in the original program was reachable from multiple definitions, the new variables are disambiguated by introducing ϕ-functions at control-flow joins. Each ϕ-function defines a new SSA variable by selecting the right SSA variable whose definition was traversed in the actual execution flow.

We require the program to be in *strict* SSA, i.e., every path from the root r to a use of a variable contains a definition of this variable. Because there is only one (static) definition per variable, strictness is equivalent to the *dominance property*, which states that each use of a variable is dominated by its definition. This is true for all uses including a use in a ϕ-operation by considering that such a use actually takes place in the predecessor block from where it originates.

3.3 Liveness

Intuitively, a variable is *live* at a program point when its value is used later by any dynamic execution. Statically, liveness can be approximated by following paths of the CFG, backwards, from the uses of a given variable to its definitions (unique definition in SSA). The variable is live at all program points along these paths. For a CFG node q, representing an instruction or a basic block, a variable `v` is *live-in* at q if there is a path, not containing the definition of `v`, from q to a node where `v` is used. It is *live-out* at q if it is live-in at some successor of q.

The computation of live-in and live-out sets at the entry and the exit of basic blocks is usually termed *liveness analysis*, i.e., algorithms operate at the granularity of blocks. It is indeed sufficient to consider only these sets since liveness within a block can be recomputed from its live-out set, either by traversing the block or by precomputing the variables *defined* and the variables *upward-exposed* in the block. A variable is upward-exposed in a block B when it is used in B and not defined earlier in B – in strict SSA, this simply means not defined in B.

However, the special behavior of ϕ-operations often causes confusion on where their operands are actually used and defined. Their liveness should be considered with care, especially when dealing with SSA destruction [11,37,5]. To make the description of algorithms easier, we follow the definition by Sreedhar [37]. For a ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ in block B_0, where a_i comes from block B_i, then:

- a_0 is live-in for B_0, but, w.r.t this ϕ-function, not live-out for B_i, $i > 0$.
- a_i, $i > 0$, is live-out of B_i, but, w.r.t this ϕ-function, not live-in for B_0.

This semantics, which corresponds to placing a copy of a_i to a_0 on each edge from B_i to B_0, can be expressed by the following *data-flow equations*:

$$\begin{aligned} \text{LiveIn}(B) &= \text{PhiDefs}(B) \cup \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) \setminus \text{Defs}(B)) \\ \text{LiveOut}(B) &= \bigcup_{S \in \text{succs}(B)} (\text{LiveIn}(S) \setminus \text{PhiDefs}(S)) \cup \text{PhiUses}(B) \end{aligned}$$

where $\text{PhiDefs}(B)$, resp. $\text{PhiUses}(B)$, denotes the variables defined, resp. used, by ϕ-operations at entry of B, resp. a successor block of B. The algorithms presented hereafter follow this semantics. They require minor modifications when other ϕ-semantics are desired. We will come back to these subtleties in Section 4.1.

3.4 Complexity of Liveness Algorithms

The running times of liveness algorithms depend on several parameters. Some can only be evaluated by experiments, e.g., the locality in data structures, the cost of function calls instead of inlined operations, etc. This is discussed in Section 5.

Complexity Parameters. Usually, liveness algorithms consider only the set W of *non-local variables*, i.e., the variables whose live-ranges cross some basic block boundary. The complexity of set operations is then measured in terms of $|W|$, the cardinality of W. However, to identify non-local variables, to identify uses and definitions, all instructions of the program P need to be visited. Traversing its internal representation is costly and not directly linked to $|W|$ as it involves all variables. Thus, any liveness algorithm requires at least $|P|$ operations to read the program and, in practice, it is better to read it only once.

After possibly some precomputations in $O(|P|)$ operations, liveness algorithms work on the CFG $G = (V, E, r)$. The number of operations can then be evaluated in terms of $|V|$ and $|E|$, i.e., the number of times blocks and control-flow edges are visited. Hereafter, we assume $|V| - 1 \leq |E| \leq |V|^2$. The costs of these operations depend on the data structures used – e.g., lists, bitsets, or sparse sets [12] – both for intermediate results (e.g., uses of a variable or upward-exposed uses in a block) and for the final results, i.e., the live-in and live-out sets. Here, we will mainly discuss the case of bitsets, as explained in Section 5.

Standard Data-Flow Approaches. The data-flow equations of Section 3.3 can be solved using a simple iterative worklist algorithm that propagates liveness information among the basic blocks of the CFG. The liveness sets are refined until a fixed point is reached. When the worklist contains CFG edges, the number

of set operations can be bounded by $O(|E||W|)$ [32], as each set can be modified (grow) at most $|W|$ times. As recalled in Section 2, another bound can be derived for the *round robin* algorithm [21,23], based on the *loop connectedness* $d(G,T)$ of the reverse CFG G, i.e., the maximal number of back edges (with respect to a depth-first spanning tree T) in a cycle-free path in G. The algorithm traverses the complete CFG on every iteration, at most $(d(G,T)+3)$ times, and thus results in $O(|E|(d(G,T)+3))$ set operations. These operations are mainly unions of sets, which can be performed in $O(|W|)$ for bitsets or ordered lists.

In addition, both need to precompute the upward-exposed uses and definitions of each basic block. This requires visiting every instruction once, thus in time $O(|P|)$ where $|P|$ is the size of the program representation. Thus, for bitsets, the overall complexity is either $O(|P|+|E||W|^2)$ or $O(|P|+|E||W|(d(G,T)+3))$ depending on the update strategy. Experiments indicate that a mixed approach combining the worklist and the round-robin principles performs best in practice [15]. In comparison, our liveness data-flow algorithm for strict SSA has complexity $O(|P|+|E||W|)$, thus is near-optimal as it includes the time to read the program, $O(|P|)$, and the time to propagate/generate the output, $O(|E||W|)$.

Path-Exploration Approaches. As Appel's path exploration [3, p. 429], the bottom-up logic approach of McAllester [29] works at the granularity of instructions, each variable considered independently. Its complexity is $O(|N||W|)$, for $|N|$ instructions, assuming that instructions have at most two successors, i.e., $|E| \leq 2|N|$, and at most two uses and one definition, thus $|P|$ is $O(|N|)$. With such assumptions, our complexity bound $O(|P|+|E||W|)$ is thus also $O(|N||W|)$. But the converse is not true if $|N|$ is not $O(|E|)$, i.e., working at the granularity of basic blocks gives a better complexity when basic blocks are large ($E \ll N$).

A direct generalization of McAllester's result to programs appearing in actual compilers – e.g., with Horn formulae expressed at the granularity of instructions and solved by the algorithm exposed by Minoux [30] – would lead to a sub-optimal complexity $O(|P||W|)$. Actually, it is important to avoid traversing the program multiple times to get $O(|P|)$ and not $O(|P||W|)$, or, even worse, a complexity that depends on the total number of variables, and not just non-local variables. In [7], we showed how to design optimized algorithms based on path exploration, operating at the basic block level, with complexity $O(|P|+|E||W|)$. Due to space limitations, we do not detail them here but we compare them to our novel data-flow algorithm in the experimental section (Section 5). In brief, compared to such path-exploration algorithms, the main interest of our loop-forest algorithm is that it operates directly on sets, i.e., all live variables at the same time, which leads to better locality and faster operations using bitsets.

4 Computing Liveness Sets Using Loop-Nesting Forests

Instead of computing a fixed point, we now show that liveness information can be derived in two passes over the blocks of the CFG by exploiting properties of strict SSA. The first version of the algorithm requires the CFG to be reducible.

We then show that arbitrary CFGs can be handled elegantly and with no additional cost, except for a cheap preprocessing step on the loop-nesting forest.

4.1 Liveness Sets on Reducible Control-Flow Graphs

The algorithm proceeds in two steps. This will be true for irreducible CFGs as well, with a slight modification described in Section 4.2. These two steps are:

1. A backward pass propagates partial liveness information, bottom up, using a postorder traversal of the CFG.
2. The partial liveness sets are then refined by traversing the loop-nesting forest, propagating liveness from loop-headers down to all basic blocks within loops.

Algorithm 1 shows the initialization to compute liveness in two passes.

Algorithm 1. Two-passes liveness analysis: reducible CFG

```
1: function Compute_LiveSets_SSA_Reducible(CFG)
2:     for each basic block B do
3:         mark B as unprocessed
4:     DAG_DFS(R)                ▷ R is the CFG root node (denoted r in Section 3.1)
5:     for each root node L of the loop-nesting forest do
6:         LoopTree_DFS(L)
```

The postorder traversal is shown by Algorithm 2, which performs a simple depth-first search and associates every basic block of the CFG with partial liveness sets. The algorithm roughly corresponds to the precomputation step of the traditional iterative data-flow analysis. Loop-edges are not considered during this traversal (Line 2). The next phase, traversing the loop-nesting forest, is shown by Algorithm 3. The live-in and live-out sets of all basic blocks in a loop are unified with the liveness sets of its loop-header. This is sufficient to compute valid liveness information because a variable whose live-range crosses a back-edge of the loop is live-in and live-out in all blocks of the loop (see Section 4.1).

Algorithm 2. Partial liveness, with postorder traversal

```
1: function DAG_DFS(block B)
2:     for each S ∈ succs(B) if (B, S) is not a loop-edge do
3:         if S is unprocessed then DAG_DFS(S)
4:     Live = PhiUses(B)          ▷ Variables used by φ-functions in B's successors.
5:     for each S ∈ succs(B) if (B, S) is not a loop-edge do
6:         Live = Live ∪ (LiveIn(S) \ PhiDefs(S))
7:     LiveOut(B) = Live
8:     for each program point p in B, backward do
9:         remove variables defined at p from Live
10:        add uses at p in Live
11:    LiveIn(B) = Live ∪ PhiDefs(B)
12:    mark B as processed
```

Algorithm 3. Propagate live variables within loop bodies

```
1: function LoopTree_DFS(node N of the loop forest)
2:     if N is a loop node then
3:         Let B_N = Block(N)                          ▷ The loop-header of N
4:         Let LiveLoop = LiveIn(B_N) \ PhiDefs(B_N)
5:         for each M ∈ LoopTree_children(N) do ▷ Visit children in the loop forest
6:             Let B_M = Block(M)                      ▷ Loop-header or block
7:             LiveIn(B_M) = LiveIn(B_M) ∪ LiveLoop
8:             LiveOut(B_M) = LiveOut(B_M) ∪ LiveLoop
9:             LoopTree_DFS(M)
```

Complexity. In contrast to iterative data-flow algorithms, our algorithm has only two phases. The first traverses the CFG once, the second traverses the loop-nesting forest once. The CFG traversal of Algorithm 2 performs $O(|V| + |E|)$ unions of sets and $O(|P|)$ set insertions. Thus, assuming $|V| - 1 \leq |E|$, the complexity of the first phase is $O(|E||W|+|P|)$ for bitsets. The size of the forest is at most twice the number of basic blocks $|V|$ in the CFG, because every loop node in the loop-nesting forest has at least one child node representing a basic block (a forest leaf). Thus, the loop-forest traversal in Algorithm 3 induces $O(|V|)$ set (union) operations. Since $|V| - 1 \leq |E|$, this phase does not change the overall complexity mentioned above. The same is true for the unmark initialization phase. Our non-iterative data-flow algorithm has thus the expected near-optimal complexity $O(|P|+|E||W|)$, as claimed before. It avoids the multiplicative factor that bounds the number of iterations in standard iterative data-flow algorithm.

Correctness. The previous algorithms were specialized for the case where ϕ-functions are interpreted as parallel copies at the preceding CFG edges. For the correctness proofs, we resort to the following, more generic, ϕ-semantics. A ϕ-function $a_0 = \phi(a_1, \ldots, a_n)$ at basic block B_0, receiving its arguments from blocks B_i, $i > 0$, is represented by a fresh variable a_ϕ, a copy $a_0 = a_\phi$ at B_0, and copies $a_\phi = a_i$ at B_i, for $i > 0$. Now, with respect to this ϕ-function, a_i, $i > 0$, is not live-out at B_i and a_0 is not live-in at B_0 anymore. As for a_ϕ, since it is not an SSA variable, it is not covered by the following lemmas. But its live-range is easily identified: it is live-in at B_0 and live-out at B_i, $i > 0$, and nowhere else. Other ϕ-semantics extend the live-ranges of the ϕ-operands with parts of the live-range of a_ϕ and can thus be handled by locally refining the live-in and live-out sets. This explains why, in Algorithm 2, PhiUses(B) is added to LiveOut(B) (Line 4), PhiDefs(B) is added to LiveIn(B) (Line 11), and PhiDefs(S) is removed from LiveIn(S) (Line 6). This ensures that the variable defined by a ϕ-function is marked as live-in and its uses as live-out at the predecessors. A similar adjustment appears on Line 4 of Algorithm 3.

The first pass propagates the liveness sets using a postorder traversal of the *reduced graph* $\mathcal{F}_\mathcal{L}(G)$, obtained by removing all loop-edges[2] from G. The

[2] Again, for a reducible CFG, the loop forest and the loop-edges are uniquely defined.

following two lemmas characterize the variables that may not be marked as live-in to a block after this pass. Detailed proofs are provided in a research report [7].

Lemma 1. *Let G be a reducible CFG,* `v` *an SSA variable, and d its definition. If L is a maximal loop not containing d, then* `v` *is live-in at the loop-header h of L iff there is a path in $\mathcal{F}_{\mathcal{L}}(G)$, not containing d, from h to a use of* `v`*.*

Lemma 2 covers the case when no loop L satisfies the conditions of Lemma 1.

Lemma 2. *Let G be a reducible CFG,* `v` *an SSA variable, and d its definition. Let p be a node of G such that all loops containing p also contain d. Then* `v` *is live-in at p iff there is a path in $\mathcal{F}_{\mathcal{L}}(G)$, from p to a use of* `v`*, not containing d.*

Algorithm 2, which propagates liveness information along the DAG $\mathcal{F}_{\mathcal{L}}(G)$, can only mark live-in variables that are indeed live-in. Moreover, if, after this propagation, a variable `v` is missing in the live-in set of a CFG node p, Lemma 2 shows that p belongs to a loop that does not contain the definition of `v`. Let L be such a maximal loop. According to Lemma 1, `v` is correctly marked as live-in at the header of L. The next lemma shows that the second pass (Algorithm 3) correctly adds variables to the live-in/live-out sets where they are missing.

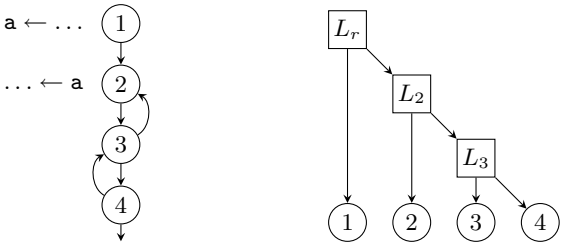
Lemma 3. *Consider a reducible CFG, L a loop, and* `v` *an SSA variable. If* `v` *is live-in at the loop-header of L, it is live-in and live-out at every CFG node in L.*

This lemma proves the correctness of the second pass, which propagates the liveness information within loops. Every CFG node, which is not yet associated with accurate liveness information, is properly updated by this second pass. Moreover, no variable is added where it should not.

Example 1. Figure 1a shows a pathological case for iterative data-flow analysis. The precomputation does not mark variable `a` as live throughout the two loops. An iteration is required for every loop-nesting level until the final solution is found. In our algorithm, after the CFG traversal, the traversal of the loop forest (Figure 1b) propagates the missing liveness information from the loop-header of L_2 within the loop's body and all inner loops, i.e., blocks 3 and 4 of L_3. □

4.2 Liveness Sets on Irreducible Control-Flow Graphs

It is well-known that every irreducible CFG can be transformed into a semantically *equivalent* reducible CFG, for example, using node splitting [22,1]. The graph may, unfortunately, grow exponentially during the processing [13]. However, when liveness information is to be computed, a relaxed notion of equivalence is sufficient. We first show that every irreducible CFG can be transformed into a reducible CFG, without size explosion, such that the liveness in both graphs is *equivalent*. Actually, there is no need to transform the graph explicitly. Instead, the effect of the transformation can be directly emulated in Algorithm 2, with a slight modification, so as to handle irreducible CFGs.



(a) Control-flow graph (b) Loop-nesting forest

Fig. 1. Bad case for iterative data-flow analysis

For every loop L, $\mathit{EntryEdges}(L)$ denotes the set of entry-edges, i.e., the edges leading, from a basic block that is not part of the loop L, to a block within L. $\mathit{Entries}(L)$ denotes the set of L's entry-nodes, i.e., the nodes that are target of an entry-edge. Similarly, $\mathit{PreEntries}(L)$ denotes the set of blocks that are the source of an entry-edge. The set of loop-edges is given by $\mathit{LoopEdges}(L)$. Given a loop L from a graph $G = (V, E, r)$, we define the graph $\Psi_L(G) = (V', E', r)$ as follows. The graph is extended by a new node δ_L, which represents the (unique) loop-header of L after the transformation. All edges entering the loop from preentry-nodes are redirected to this new header. The loop-edges of L are similarly redirected to δ_L and additional edges are inserted leading from δ_L to L's former loop-headers. More formally:

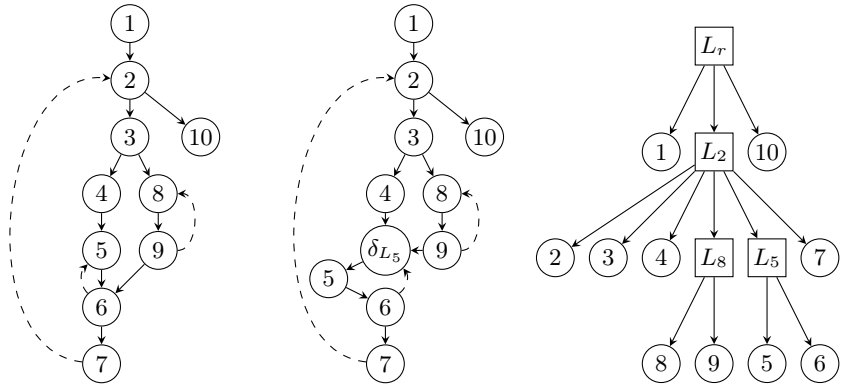
$$E' = E \setminus \mathit{LoopEdges}(L) \setminus \mathit{EntryEdges}(L) \cup \{(s, \delta_L) \mid s \in \mathit{PreEntries}(L)\}$$
$$\cup \{(s, \delta_L) \mid \exists (s, h) \in \mathit{LoopEdges}(L)\} \cup \{(\delta_L, h) \mid h \in \mathit{LoopHeaders}(L)\}$$

Repeatedly applying this transformation yields a reducible graph, slightly larger than the original graph, in which each node is still reachable from the root r. Depending on the order in which loops are considered, entry-edges may be updated several times during the processing in order to reach their final positions. But the loop-forest structure remains the same.

Ramalingam proposed a similar transformation [35, p. 473], which is intended to build an acyclic graph while preserving dominance. It is easy to see that his transformation does *not* preserve liveness and is thus not suited for our purpose.

Example 2. Figure 2c shows a loop forest for the CFG of Figure 2a, where node 5 was selected as loop-header for L_5, the loop containing the nodes 5 and 6. As both nodes are entry-nodes, via the preentry-nodes 4 and 9, the CFG is irreducible. The transformed reducible graph $\Psi_{L_5}(G)$ in Figure 2b might not reflect the semantics of the original program during execution, but it preserves the liveness of the original CFG, for a strict SSA program, as Theorem 1 will show. □

To avoid building this transformed graph explicitly, an elegant alternative is to modify the CFG traversal (Algorithm 2). To make things simpler, we assume



(a) G: Irreducible (b) $\Psi_L(G)$: Reducible (c) A loop-nesting forest

Fig. 2. Transformation of an irreducible CFG using a loop-nesting forest

that the loop forest is built so that, as in Havlak's loop forest construction [19], each loop L has a single[3] loop-header, which can thus implicitly be fused with δ_L. It is then easy to see that, after all CFG transformations, an entry-edge (s, t) is redirected from s to $\mathsf{HnCA}(s, t)$ the loop-header of the *highest non common ancestor* of s and t, i.e., of the highest ancestor of t in the loop forest that is not an ancestor of s. Thus, whenever an entry-edge (s, t) is encountered during the traversal, we just have to visit $\mathsf{HnCA}(s, t)$ instead of t, i.e., to visit the representative of the largest loop containing the edge target, but not its source. To perform this modification, we replace all occurrences of S by $\mathsf{HnCA}(B, S)$ at Lines 3 and 6 of Algorithm 2, in order to handle irreducible flow graphs.

Complexity. The changes to the original forest algorithm are minimal and only involve the invocation of HnCA to compute the highest non common ancestor. This function solely depends on the structure of the loop-nesting forest of G. Assuming that HnCA is precomputed, the complexity results obtained previously still hold as the number of edges $|E|$ does not change. The highest non common ancestors can easily be derived by propagating sets of basic blocks from the leaves upwards to the root of the loop-nesting forest using a depth first search. This enumerates all basic block pairs exactly once at their respective least common ancestor. Since the overhead of traversing the forest is negligible, the worst case complexity can be bounded by $O(|V|^2)$. More involved algorithms, as for the lowest common ancestor problem [4], are possible, which process the tree in $O(|V|)$, so that subsequent HnCA queries may be answered in constant time per query. In other words, modifying the algorithm with HnCA to handle irreducible CFGs does not change the overall complexity.

[3] To handle loop forests with loops having several loop-headers, we can select one particular loop-header to be *the* loop representative (B_N in Algorithm 3). But then we need to add edges from this loop-header to any other loop-header.

Correctness. We now prove that, in strict SSA, the liveness of the resulting reducible CFG is equivalent to the liveness of the original one. The following results hold even for a loop forest whose loops have several loop-headers. First, to be able to apply the lemmas and algorithms of Section 4 to the reducible CFG $\Psi_L(G)$, we prove that any definition of a variable still dominates its uses.

Lemma 4. *If d dominates u in G, d dominates u in $\Psi_L(G)$.*

It remains to show that, for every basic block present in both graphs, the live-in and live-out sets are the same. This is proved by the following theorem.

Theorem 1. *Let* v *be an SSA variable, G a CFG, transformed into $\Psi_L(G)$ when considering a loop L of a loop forest of G. Then, for each node q of G,* v *is live-in (resp. live-out) at q in G iff* v *is live-in (resp. live-out) at q in $\Psi_L(G)$.*

5 Experiments

As previously shown, the theoretical complexity of our non-iterative liveness algorithm is near-optimal: it includes the time to read the program, $O(|P|)$, and the time to propagate/generate the output, $O(|E||W|)$. Moreover, variables are added to liveness sets only when actually needed. However, it is still important to evaluate the runtime of the algorithm in practice. We thus compared our algorithm with state-of-the-art approaches, namely an optimized iterative data-flow algorithm, following the "single stack worklist" approach of Cooper et al. [15], and two variants based on path exploration (see end of Section 3.4). The first variant, called use-by-use, traverses the program backwards and, for every encountered variable use, starts a backward depth-first search to find the variable's definition. The variable is added to the live-in and live-out sets along the discovered paths. The other variant, called var-by-var, processes one variable after the other and relies on precomputed def-use chains to find the variable's uses. Both variants are optimized to exploit SSA properties, achieving the near-optimal complexity $O(|P| + |E||W|)$ – for details consult the accompanying report [7].

The algorithms were implemented using the production compiler for the STMicroelectronics ST200 VLIW family, based on GCC as front-end, the Open64 optimizers, and the LAO code generator [17]. We computed liveness relatively late during code generation in the LAO optimizer, shortly before prepass scheduling. For this evaluation, we used bitsets to represent liveness sets, which offer faster accesses, but are often considered to be less efficient in terms of memory consumption and are expected to degrade in performance as the number of variables increases, due to more cache misses and memory transfers. We also investigated the use of ordered pointer-sets, which promise reduced memory consumption at the expense of rather costly set operations. But, as our experiments indicate that bitsets are overall superior [7], we limit our discussion to bitsets.

We applied the various algorithms to the C programs of the SPECINT 2000 benchmark suite to measure the time to compute liveness information. To obtain reproducible results, the execution time was measured using the instrumentation

and profiling tool *callgrind*. These measurements include the number of instructions executed and the memory accesses via caches. Using them, a cycle estimate is computed for the liveness computation only, which minimizes the impact of other compiler components and other running programs on the measurements. As the number of non-local variables depends largely on the compiler optimizations performed before liveness calculation, we investigated the behavior of the various algorithms for optimized and unoptimized programs using the compiler flags `-O2` and `-O0` respectively. All measurements are given relative to the iterative data-flow approach, which performed the worst in all our experiments.

Since most variables are kept in memory at optimization level `-O0`, the number of non-local variables is low (at most 19 in our experiments) and their live-ranges are short. The results (see [7] for details) thus mainly reveal the intrinsic overhead of the different implementations, including artifacts stemming from the host compiler and the host machine. The var-by-var algorithm, which simply iterates over the small set of non-local variables, performs best, as it is the least impacted by the number of basic blocks and operations in the program. The measurements account for the precomputation of the def-use chains, which appears to be less costly than the explicit traversal in the use-by-use algorithm. Our loop-forest algorithm cannot reach the performances of the two path-exploration solutions, which show an average speed-up of 1.80 for the var-by-var algorithm and 1.63 for the use-by-use variant (2.19 and 1.99 compared to iterative data-flow). However, we already observe a speed-up of 1.22 on average in comparison to the state-of-the-art iterative data-flow analysis.

The characteristics of optimized programs are different. The structure of the live-ranges is more complex and the liveness sets are larger. Table 1 shows the number of non-local variables, basic blocks, and operations for the optimized benchmarks. For such programs, the iterative data-flow analysis is still the worst but, now, the var-by-var algorithm is performing worse than the two others, see Figure 3. Our loop-forest approach clearly outperforms both

Table 1. Characteristics of optimized programs

	# Variables			# Blocks			# Operations		
Benchmark	min	avg	max	min	avg	max	min	avg	max
164.gzip	11	104	586	2	32	212	22	226	1312
175.vpr	10	84	573	2	33	492	21	224	1734
176.gcc	10	119	36063	2	37	1333	11	282	41924
181.mcf	12	52	118	2	18	52	24	135	439
186.crafty	11	147	1048	2	67	2112	22	547	9836
197.parser	10	58	1076	2	21	343	21	126	1942
253.perlbmk	10	61	1947	2	28	731	16	180	4876
254.gap	10	95	6472	2	31	778	13	244	9169
255.vortex	10	51	645	2	26	667	21	166	3361
256.bzip2	10	73	972	2	22	282	21	163	1931
300.twolf	10	186	3659	2	53	715	12	458	8691

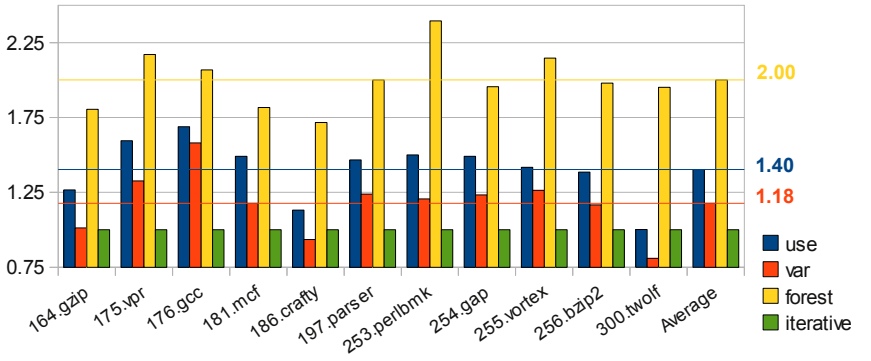


Fig. 3. Speed-up relative to iterative data-flow on optimized programs, with bitsets

path-exploration algorithms, with, on average, speed-ups of 1.69 (= 2.00/1.18) and 1.43 (= 2.00/1.40) respectively. This is explained by the relative cost of the fast bitset operations, in particular set unions, in comparison to the cost of traversing the CFG. Furthermore, the locality of memory accesses becomes a relevant performance factor. Both the use-by-use and our loop-forest algorithms operate *locally* on the bitsets surrounding a given program point. The inferior locality, combined with the necessary precomputation of the def-use chains, explains the poor results of the var-by-var approach in this experimental setting.

More detailed results in comparison to the iterative data-flow approach are given in [7] on a per-module basis, using one data point for every source file. The loop-forest and the use-by-use algorithms on average outperform the iterative one by a factor of 2 and 1.5. Some extreme cases, showing speed-ups by a factor higher than 8, are caused by unusual, though relevant, loop structures in source code generated by the parser generator *bison* (`c-parse.c` of `gcc`, `perly.c` of `perlbmk`), which increase the number of iterations of the iterative algorithm. On the other hand, all cases where the iterative algorithm outperforms the other algorithms are due to implementation artifacts: the analyzed functions do not contain any non-local variables thus slight variations in the executed code, its placement, and the cache state become relevant. The var-by-var approach is often even slower than the iterative one but on average shows a speed-up of 1.18.

Our new algorithm spends most of the time on the backward phase, which amounts to 68% and 53% of the analysis time for unoptimized and optimized programs respectively. The forward phase is almost negligible and contributes only 3% and 6% respectively (with a maximum of 19%). Setting up and initializing data structures, by contrast, takes a large share of 27% and 36% respectively. It is interesting to see that this share increases for optimized programs, due to the large number of variables. The time to construct the loop forest was excluded from these statistics, because loop information is needed for other optimizations anyway, e.g., register allocation, code motion, if-conversion. The naive (quadratic) loop-forest construction available in our framework takes only about 15% of the time of our new liveness algorithm, for optimized programs.

6 Conclusion

Liveness analysis is the basis for many compiler optimizations. However, code transformations often invalidate this information, which has to be repeatedly recomputed. Fast algorithms are thus required to minimize its overhead.

This work proposes an improvement to the traditional iterative data-flow analysis for programs in strict SSA form, which consists of *only two phases*. The first phase resembles the precomputation phase of the standard approach providing partial liveness sets. The second pass replaces the traditional iterative refinement by a *single* traversal of a loop-nesting forest of the control-flow graph.

Our algorithm has the same theoretical complexity as optimized techniques based on path exploration that we developed for comparison. But it operates directly on sets, i.e., all live variables at the same time, and thus is more likely to offer better locality and faster operations using bitsets. As our experiments show, our loop-forest algorithm outperforms the iterative method by a factor of 2 on average for the SPECINT 2000 benchmark suite. Also, for optimized codes, having a large number of non-local variables and complex control flow, our loop-forest approach outperforms by a factor at least 1.43 the path-exploration techniques we proposed, whereas, for unoptimized codes, having very few non-local variables, the path-exploration algorithms appear to be suited best.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (2006)
2. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Communications ACM 19(3), 137 (1976)
3. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press (2002)
4. Bender, M., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
5. Boissinot, B., Darte, A., Dupont de Dinechin, B., Guillon, C., Rastello, F.: Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In: Symp. on Code Generation and Optimization (CGO 2009), pp. 114–125. ACM (2009)
6. Boissinot, B.: Towards an SSA-Based Compiler Back-End: Some Interesting Properties of SSA and its Extensions. PhD thesis, ENS-Lyon (September 2010)
7. Boissinot, B., Brandner, F., Darte, A., de Dinechin, B.D., Rastello, F.: Computing liveness sets for SSA-form programs. TR Inria RR-7503 (2011)
8. Boissinot, B., Brisk, P., Darte, A., Rastello, F.: SSI properties revisited. ACM Transactions on Embedded Computing Systems, Special Issue on Software and Compilers for Embedded Systems (2009) (to appear)
9. Boissinot, B., Hack, S., Grund, D., de Dinechin, B.D., Rastello, F.: Fast liveness checking for SSA-programs. In: IEEE/ACM Symposium on Code Generation and Optimization (CGO 2008), pp. 35–44. ACM (2008)
10. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register Allocation: What does the NP-Completeness Proof of Chaitin et al. Really Prove? In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 283–298. Springer, Heidelberg (2007)

11. Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.: Practical improvements to the construction and destruction of static single assignment form. Software—Practice and Experience 28(8), 859–881 (1998)
12. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems (LOPLAS) 2, 59–69 (1993)
13. Carter, L., Ferrante, J., Thomborson, C.: Folklore confirmed: Reducible flow graphs are exponentially larger. In: ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL 2003), pp. 106–114. ACM (2003)
14. Click, C., Paleczny, M.: A simple graph-based intermediate representation. In: ACM Workshop on Intermediate Representations, pp. 35–49. ACM (1995)
15. Cooper, K.D., Harvey, T.J., Kennedy, K.: An empirical study of iterative data-flow analysis. In: 15th International Conference on Computing (ICC 2006), pp. 266–276. IEEE Computer Society, Washington, DC, USA (2006)
16. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadek, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
17. de Dinechin, B.D., de Ferrière, F., Guillon, C., Stoutchinin, A.: Code generator optimizations for the ST120 DSP-MCU core. In: Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2000), pp. 93–102. ACM (2000)
18. Hack, S., Grund, D., Goos, G.: Register Allocation for Programs in SSA Form. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 247–262. Springer, Heidelberg (2006)
19. Havlak, P.: Nesting of reducible and irreducible loops. ACM Transactions on Programming Languages and Systems 19(4), 557–567 (1997)
20. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. Journal of the ACM 21(3), 367–375 (1974)
21. Hecht, M.S., Ullman, J.D.: Analysis of a simple algorithm for global data flow problems. In: 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1973), pp. 207–217. ACM (1973)
22. Janssen, J., Corporaal, H.: Making graphs reducible with controlled node splitting. ACM Trans. on Programming Languages and Systems 19, 1031–1052 (1997)
23. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. Journal of the ACM 23(1), 158–171 (1976)
24. Kildall, G.A.: A unified approach to global program optimization. In: Symposium on Principles of Programming Languages (POPL 1973), pp. 194–206. ACM (1973)
25. Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D.: Design of the Java HotSpot™ client compiler for Java 6. ACM Transactions on Architecture and Code Optimization 5, 1–32 (2008)
26. Leung, A., George, L.: Static single assignment form for machine code. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 999), pp. 204–214. ACM Press (1999)
27. LibFirm: A library that provides an intermediate representation and optimisations for compilers, http://pp.info.uni-karlsruhe.de/firm
28. LLVM: The LLVM compiler infrastructure, http://llvm.org
29. McAllester, D.: On the complexity analysis of static analyses. Journal of the ACM 49, 512–537 (2002)
30. Minoux, M.: LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. Inform. Processing Letters 29, 1–12 (1988)
31. Mono: NET development framework, http://www.mono-project.com
32. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc. (1999)

33. Pereira, F.M.Q., Palsberg, J.: Register Allocation Via Coloring of Chordal Graphs. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 315–329. Springer, Heidelberg (2005)
34. Ramalingam, G.: Identifying loops in almost linear time. ACM Transactions on Programming Languages and Systems 21(2), 175–188 (1999)
35. Ramalingam, G.: On loops, dominators, and dominance frontiers. ACM Transactions on Programming Languages and Systems 24(5), 455–490 (2002)
36. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Computing Surveys 18(3), 277–316 (1986)
37. Sreedhar, V.C., Ju, R.D.-C., Gillies, D.M., Santhanam, V.: Translating out of Static Single Assignment Form. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 194–210. Springer, Heidelberg (1999)
38. Tarjan, R.E.: Testing flow graph reducibility. Journal of Computer and System Sciences 9(3), 355–365 (1974)
39. Wimmer, C., Franz, M.: Linear scan register allocation on SSA form. In: Symp. on Code Generation and Optimization (CGO 2010), pp. 170–179. ACM (2010)

SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA

Yulei Sui[1], Sen Ye[1], Jingling Xue[1], and Pen-Chung Yew[2]

[1] School of Computer Science and Engineering, UNSW, Australia
[2] Department of Computer Science and Engineering, University of Minnesota, USA

Abstract. We present a new SPAS (*S*calable *PA*th-*S*ensitive) framework for resolving points-to sets in C programs that exploits recent advances in pointer analysis. SPAS enables intraprocedural path-sensitivity to be obtained in flow-sensitive and context-sensitive (FSCS) techniques scalably, by using BDDs to manipulate program paths and by performing pointer analysis level-by-level on a full-sparse SSA representation similarly as the state-of-the-art LevPA (the FSCS version of SPAS). Compared with LevPA using all 27 C benchmarks in SPEC CPU2000 and CPU2006, SPAS incurs 18.42% increase in analysis time and 10.97% increase in memory usage on average, while guaranteeing that all points-to sets are obtained with non-decreasing precision.

1 Introduction

There have been great advances in pointer analysis performed flow-sensitively [9,10,14], context-sensitively [19,16,17] or with both combined [23,11]. As reported recently [10,11,23], insensitive analysis can be leveraged to bootstrap sensitive analysis, thereby leading to significant improvements in scalability and precision. In particular, it is shown by LevPA [23] that flow-sensitive and context-sensitive (FSCS) analysis becomes substantially more scalable when performed on full-sparse SSA, level by level (in order of their decreasing *points-to levels*), with each level being analyzed with an inclusion-based flow-insensitive algorithm. However, little progress [20,15] has been made when path-sensitivity is also considered. Exploiting recent advances in FSCS pointer analysis, we describe a SPAS (*S*calable *PA*th-*S*ensitive) framework that enables intraprocedural path-sensitivity to be obtained scalably for C programs on top of the state-of-the-art LevPA (the FSCS version of SPAS). SPAS obtains all points-to sets with non-decreasing precision by adding small analysis overhead in both time and space, as validated using SPEC CPU2000 and SPEC2006.

Equipped with path-sensitivity, a FSCS pointer analysis is equally as or more precise, as illustrated in Figure 1. With such path-sensitive precision, the quality of many software tools and techniques in program optimization, analysis and verification can be significantly improved. Examples include bug hunting [13], memory leak analysis [20] and software vulnerability detection [15,21].

A major hindrance to path-sensitive pointer analysis is the lack of scalability. We tackle it by tracking program paths using Binary Decision Diagrams (BDDs)

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 155–171, 2011.

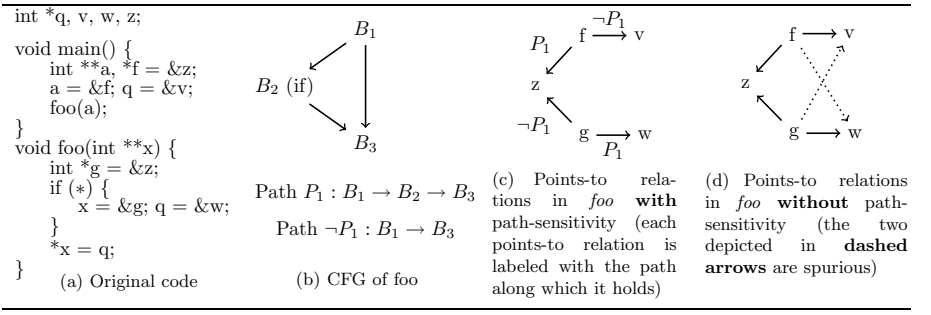


Fig. 1. Effects of path-sensitivity on indirect updates at a store $*x{=}q$ in FSCS analysis

and by infusing path-sensitivity into FSCS pointer analysis seamlessly on full-sparse SSA. Our generalization is simple, drops easily on a full-sparse FSCS pointer analyzer like LevPA [23], and retains the scalability of the underlying FSCS analysis (by adding small overhead in analysis time and memory usage).

Presently, SPAS captures path correlation only without ruling out infeasible paths. In summary, this paper makes the following contributions:

- SPAS is the first *full-sparse* path-sensitive FSCS pointer analysis;
- SPAS is the first to encode program paths using BDDS on full-sparse SSA;
- SPAS obtains (intraprocedural) path-sensitivity efficiently on full-sparse SSA, level by level, with each level being analyzed flow-insensitively; and
- SPAS has compatible performance as the state-of-art FSCS analyzer LevPA (the FSCS-version of SPAS). With both implemented in Open64, SPAS adds small average overhead (18.42% more in time and 10.97% more in space) for all the 27 C programs in SPEC CPU2000 and CPU2006, while ensuring that all points-to sets are obtained with non-decreasing precision. To the best of our knowledge, SPAS is the fastest path-sensitive FSCS pointer analysis for C reported in the literature.

2 Related Work

There is not much reported on performing whole-program pointer analysis flow-, context- and path-sensitively. We review some solutions using sparse SSA and BDDs and some limited amount of prior work on path-sensitive pointer analysis.

Sparsity. Unlike iterative dataflow-based pointer analysis [12,3,7], SSA-based pointer analysis [9,10,23] is sparse and thus more scalable, as SSA allows points-to information to flow directly from variable definitions to their uses only.

In [9], Hardekopf and Lin presented a semi-sparse flow-sensitive analysis. By putting *top-level pointers* in SSA, their def-use information can be exposed directly. Lately [10], they generalized their work by making it full-sparse. This is done by using a flow-insensitive inclusion-based pointer analysis to compute the required def-use information in order to build SSA for all variables. However,

their algorithms are not context-sensitive. Yu et al. [23] introduced LevPA for performing FSCS pointer analysis on full-sparse SSA. In LevPA, points-to resolution and SSA construction are performed together, level by level, in decreasing order of their points-to levels. Our SPAS framework is a scalable generalization of FSCS pointer analyzers like LevPA to obtain intraprocedural path-sensitivity.

BDDs. Berndl et al. [1] proposed to use BDDs to encode points-to relations and also studied the impact of BDD variable ordering on analysis performance. Later, BDDs were also used to encode transfer functions [24] and contexts [19,23] in context-sensitive analysis. In [21], Xie and Aiken discussed to use BDDs to represent program paths and simplify paths heuristically in SATURN, a SAT-based bug detection tool. In our SPAS pointer analyzer, BDDs are used to encode paths (and contexts) on full-sparse SSA using a standard BDD library. This enables the pointers to be resolved using a guarded inclusion-based insensitive analysis on full-sparse SSA, where the guards are contexts and program paths.

Path-Sensitivity. We are unaware of any prior FSCS pointer analyzer supporting path-sensitivity on sparse SSA. In some bug-hunting tools like Prefix [2], path-sensitivity was exploited to look for bugs in some selected paths. A recent sound and complete generalization for SATURN [6] can compute various program properties using a SAT solver with even interprocedural path-sensitivity. However, it is not suited for whole-program points-to resolution.

SPAS and bug analysis aim to achieve different goals. Some bug detection tools rule out some infeasible paths based on branch conditions to reduce false positives but SPAS presently does not. However, SPAS can already provide more precise points-to information to make these tools more effective. Finally, Gutzmann et al. [8] discussed how to filter out spurious points-to relations flowing out of a branch node but their approach neither captures path correlation as SPAS does nor rules out infeasible paths.

3 The Basic Idea

SPAS analyzes all features of C by considering four types of assignments: $x = y$ (*copy*), $x = \&y$ (*address*), $*x = y$ (*store*) and $x = *y$ (*load*). SPAS is field-insensitive for arrays (by not distinguishing array elements) but field-sensitive for structs (by flattening and replacing them with separate variables, one for each field). SPAS names abstraction heap objects by their allocation sites.

The following sets and functions are used in some definitions given below.

- $\mathbb{O}$: set of abstract memory locations representing the variables in a program.
- $\mathcal{L} : \mathbb{O} \rightarrow \{0, \ldots, L\}$: a *level map* giving each variable a points-to level. If q may be modified by operations on p (possibly indirectly), then $\mathcal{L}(p) \geq \mathcal{L}(q)$.
- $\mathbb{C}$: set of contexts represented as Boolean expressions over the set $\mathbb{O} \times \mathbb{O}$ of points-to relations. The notation $p \rightarrow o$ means that p may point to o.
- $\mathbb{D}$: set of paths represented as Boolean conditions over the set of *decision variables* (encoding the branches in a CFG to be introduced in Section 5.1).
- $\mathcal{C} : \mathbb{C} \times \mathbb{D}$: set of combined context and path conditions used to specify under which condition in $\mathcal{C}$ a pointer may point to a memory object in $\mathbb{O}$.

Definition 1 (Formal-Ins). *Given a method m, $\mathbb{F}^m_{\text{in}} \subseteq \mathbb{O}$ denotes the set of its* formal-ins*, i.e., its formal parameters and non-local variables accessed in m.*

Definition 2 (Formal-Outs). *Given a method m, $\mathbb{F}^m_{\text{out}} \subseteq \mathbb{O}$ denotes the set of its* formal-outs*, i.e., its* return parameter *(a special local variable of m containing its return value) and non-local variables that may be modified in m.*

In flow-sensitive analysis, we speak of the points-to sets of a pointer p at program points, which are identified by the versions of p at those points in SSA.

Definition 3 (Points-to Sets). *The points-to set of a pointer p at a program point, $\mathsf{PtrSet}(p) \subseteq \mathbb{O}$, is a set of locations in $\mathbb{O}$ possibly pointed to by p.*

SPAS achieves context-sensitivity by traversing the call graph of a program bidirectionally. During a bottom-up traversal, the points-to sets of a pointer p at a point may be related to those of some formal-ins in terms of points-to maps (Definition 4). During a top-down traversal, p is resolved once the points-to sets of all its dependent formal-ins have been found (Definition 3).

Definition 4 (Points-to Maps). *The points-to map of a pointer p at a program point in a method m is given by:*

$$\mathsf{PtrMap}(p) = (\mathsf{Loc}(p), \mathsf{Dep}(p)) \tag{1}$$

where $\mathsf{Loc}(p) \subseteq \mathbb{O} \times \mathcal{C}$ contains each tuple (v, C_v, P_v) such that p may point to v in context C_v along path P_v within method m and $\mathsf{Dep}(p) \subseteq \mathbb{F}^m_{\text{in}} \times \mathcal{C}$ contains each tuple (f, C_f, P_f) such that p may point to what formal-in f points to in context C_f along path P_f within method m.

SPAS is intraprocedurally path-sensitive. Thus, the transfer functions (*MOD* and *USE*) of a formal-out in a method m are predicated by m's calling contexts (without m's path conditions). Similarly, the path conditions in their specified side effects are also ignored (and hence, the $*$'s). To support strong updates, we distinguish MAY-DEFs and MUST-DEFs at stores.

Definition 5 (*MOD*). *The transfer function MOD of a formal-out $f_{\text{out}} \in \mathbb{F}^m_{\text{out}}$ in a method m describes its interprocedural modification side effect:*

$$MOD(m, f_{\text{out}}) = (\mathsf{Loc}(f_{\text{out}}), \mathsf{Dep}(f_{\text{out}}), C^{\text{may}}_{f_{\text{out}}}, C^{\text{must}}_{f_{\text{out}}}) \tag{2}$$

indicating that f_{out} may be modified, i.e., MAY-DEF'ed in context $C^{\text{may}}_{f_{\text{out}}} \in \mathbb{C}$ to point to either (a) v for each $(v, C_v, *) \in \mathsf{Loc}(f_{\text{out}})$ when C_v holds or (b) whatever f points to for each $(f, C_f, *) \in \mathsf{Dep}(f_{\text{out}})$ when C_f holds. If $C^{\text{must}}_{f_{\text{out}}} \in \mathbb{C}$ also holds, then the MAY-DEF is actually a MUST-DEF.

Definition 6 (*USE*). *The transfer function USE of a formal-in $f_{\text{in}} \in \mathbb{F}^m_{\text{in}}$ in a method m describes its MAY-USE, i.e., interprocedural read side effect:*

$$USE(m, f_{\text{in}}) = (\mathsf{PtrSet}(f_{\text{in}}), C_{f_{\text{in}}}) \tag{3}$$

indicating that what is pointed to by f_{in} may be read in context $C_{f_{\text{in}}} \in \mathbb{C}$. (If m is a formal parameter, it does not need a USE function as m is local.)

The basic idea behind SPAS is simple. The pointers are resolved in order of their decreasing points-to levels by maintaining the invariant stated below.

*Property 1 (***Level-Wise Invariant).** Just before level ℓ is analyzed, (a) all (direct and indirect) accesses to the pointers at higher levels are in SSA, with the indirect accesses via pointer dereferencing and calls being expressed using μ (MAY-USE) and χ (MAY/MUST-DEF) operations [4], (b) all pointers at higher levels have been soundly resolved, and (c) all indirect accesses made by dereferencing the pointers at level ℓ are exposed using μ and χ operations.

The analysis performed at level ℓ is to ensure that this invariant holds at the beginning of $\ell - 1$. This is done by traversing the call graph of a program first bottom-up and then top-down iteratively. During bottom-up analysis, SPAS analyzes each method m by (B1) building its SSA (doable as Property 1(c) holds for ℓ) and (B2) computing the points-to maps for its pointers at ℓ (Definition 4). Prior to (B1), SPAS inserts μ and χ operations for each of its call sites, c, to expose the MAY-USEs and MAY/MUST-DEFs made by c's callees to every pointer at ℓ. This is done by applying the callees' transfer functions at call site c. After (B2) is done, the transfer functions of method m are computed. During top-down analysis, SPAS (T1) resolves the points-to sets of the pointers at ℓ by propagating the dependent points-to sets to formal-ins (Definition 3) and (T2) annotates the dereferences to these pointers with μ and χ operations.

4 A Motivating Example

We describe how SPAS improves FSCS pointer analysis by refining Figure 1(d) into Figure 1(c). This program may look complex but it appears to be one of the smallest examples that we can come up with in order to illustrate all key aspects of SPAS. The variables are partitioned into three levels: $\{a, x\}$ at level 2, $\{q, f, g\}$ at level 1 and $\{v, w, z\}$ at level 0. We examine the top two levels only.

Level 2. The pointers a and x are considered. The input is Figure 1(a), for which Property 1 holds trivially at this level. The output is given in Figure 2(a).

- **Bottom-Up.** When *foo* is analyzed, all accesses to x are first put in SSA. During points-to resolution, the points-to maps for its three definitions are found. In particular, x_0 is recorded to point to what formal-in x points to. By analyzing the ϕ node for x path-sensitively, x_2 is found to point to g in context true (i.e., any context) along path P_1 and what x points to in context true along $\neg P_1$. When *main* is analyzed, the points-to map of a_0 is found.
- **Top-Down.** In *main*, a_0 has been resolved locally. Binding a_0 with formal-in x at the call site to *foo* reveals that $\mathsf{PtrSet}(x_0) = \{f\}$. When *foo* is processed, the MAY-DEF to f (g) via $*x_2$ is exposed by a χ operation, where context condition $x \rightarrow f$ (true) indicates that the MAY-DEF occurs when formal parameter x points to f (in any context). Like LevPA, SPAS uses points-to relations holding at a call site to represent and distinguish calling contexts.

Level 1. The pointers q, f and g are considered. The input is Figure 2(a), for which Property 1 holds at this level. The output is given in Figure 2(b).

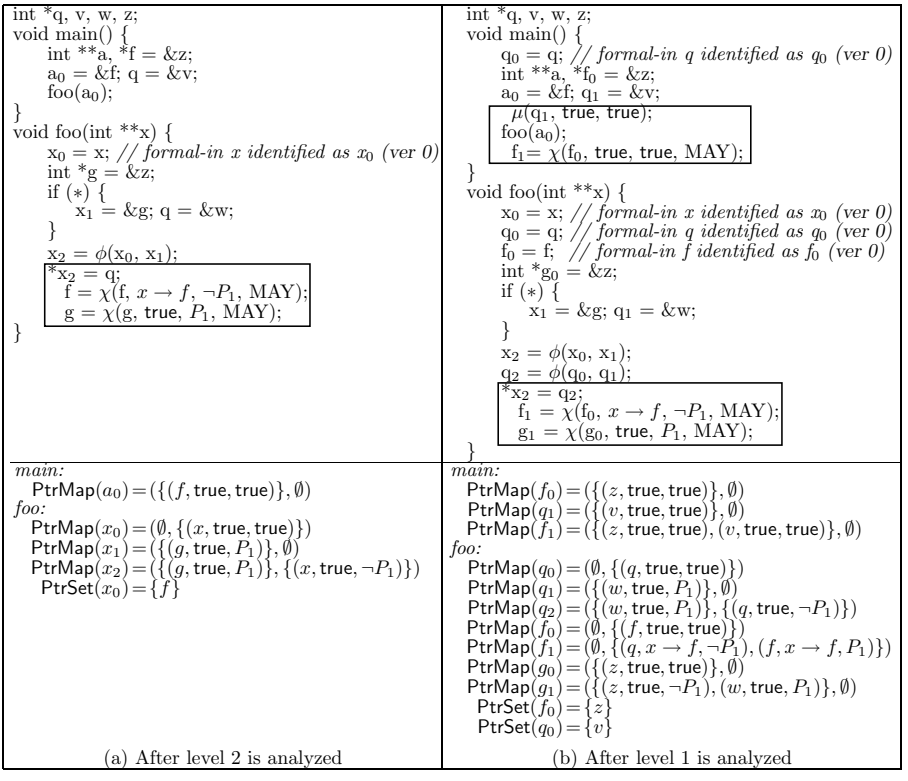


Fig. 2. Level-wise SSA construction and path-sensitive pointer analysis for Figure 1(a)

– **Bottom-Up.** When *foo* is analyzed, all accesses to the three pointers (including the two MAY-DEFs) are first put in SSA. Like x, $q_0 = q$ and $f_0 = f$ are inserted for the formal-ins q and f. Here, q is a global and f is an invisible [12] accessed via pointer dereferening. The points-to resolution for q is done similarly as x with the points-to maps obtained as shown. We now consider how store $*x_2 = q_2$ is analyzed together with its two MAY-DEFs to resolve f_1 and g_1. By capturing path correlation, SPAS deduces that f_1 points to whatever formal-in q does along path $\neg P_1$ and whatever formal-in f does along P_1 and that g_1 points to z along $\neg P_1$ and w along P_1. The transfer functions relevant for computing the side effects of the call to *foo* are:

$$\begin{aligned} MOD(foo, f) &= (\{\mathsf{Loc}(f_1), \mathsf{Dep}(f_1), x \rightarrow f, \mathsf{false}) \\ USE(foo, q) &= (\mathsf{PtrSet}(q_0), \mathsf{true}) \end{aligned} \tag{4}$$

When *main* is analyzed, a MAY-DEF for f is added since context $C_f^{\text{may}} = x \rightarrow f$ in $MOD(foo, f)$, once mapped to $a \rightarrow f$ at the call site, holds. However, this is not a MUST-DEF since $C_f^{\text{must}} = \mathsf{false}$. In addition, a MAY-USE for q is added. Then the SSA form is built. Finally, by performing the points-to resolution for f and q with the modification side effects of *foo* on f being accounted for, we obtain their points-to maps as shown.

– **Top-Down.** When *main* is analyzed, its pointers at this level are already resolved. Propagating the points-to sets of q_1 and f_0 at the call site to *foo*, we find $\mathsf{PtrSet}(f_0) = \{z\}$ and $\mathsf{PtrSet}(q_0) = \{v\}$ in *foo*. When *foo* is processed next, all its pointers at this level can be resolved by Definition 4. As a result, the points-to relations for f_1 and g_1 are obtained as shown in Figure 1(c).

SPAS obtains such improved precision with a slight increase in analysis overhead. There are several reasons for SPAS to achieve this level of scalability:

Program Paths Manipulated as BDDs. Like contexts [23], program paths are also represented and operated on using BDDs in a compact and canonical fashion, resulting in fast operations on program paths.

Full-Sparse SSA (at All Points-to Levels). As in LevPA [23], pointers are resolved, level by level, in order of their decreasing points-to levels. At the same time, the full-sparse SSA form is being built incrementally. The points-to relations of a pointer at a particular level cannot be propagated to lower-level pointers unless it has fully been resolved. Thus, the number of repropagations is reduced, leading to faster convergence for the points-to resolution.

Flow-Insensitive Points-to Resolution. SSA is ideal for enabling sparse analysis because it makes def-use information explicit. Like contexts [23], program paths are also used to guard what points-to information can be propagated across an pointer assignment. Thus, our path-sensitive pointer analysis is sped up with a guarded inclusion-based flow-insensitive pointer analysis.

5 The SPAS Framework

SPAS is a summary-based FSCS pointer analyzer with intraprocedural path-sensitivity being supported. In particular, SPAS builds *MOD* and *USE* functions for each method and applies them to all its calling contexts.

Section 5.1 discusses how to encode program paths using BDDs. Section 5.2 examines the χ and μ operators added to the classic SSA form. To ease understanding, we introduce SPAS in two stages. In Section 5.3, we focus on capturing path correlation without performing strong updates. In Section 5.4, we discuss briefly but precisely how to extend it to perform path-sensitive strong updates.

5.1 Encoding Program Paths as BDDs

SPAS does not presently distinguish the paths inside a loop-induced cycle but can analyze the first few iterations of a loop path-sensitively via loop peeling.

All branch nodes are assumed to be binary. We use *decision variables* to encode branch nodes to express program paths. The edges and blocks in a CFG (with cycles collapsed) are associated with paths as follows. The path for the incoming edge of the entry block is initialized to true (representing the set of all paths). Let B be a block with n incoming edges associated with paths $P_1, \ldots, P_n$. The path for B is $P_1 \vee \cdots \vee P_n$. If B is a branch node encoded with decision

variable Q, the paths for its two outgoing edges are $(P_1 \vee \cdots \vee P_n) \wedge Q$ and $(P_1 \vee \cdots \vee P_n) \wedge \neg Q$, respectively. Otherwise, its unique outgoing edge is $P_1 \vee \cdots \vee P_n$.

Our BDD encoding has three advantages. First, the number of BDD variables used is kept to a minimum. Second, it plays up the strengths of BDDs by exposing opportunities for path redundancy elimination. Third, the paths combined at a join node are effectively simplified (e.g., with $P_1 \vee \neg P_1$ being reduced into true), resulting in fast propagation of path conditions during points-to resolution.

5.2 Extended SSA Form

The classic SSA representation [5] is mainly useful for scalars without aliases. Following [23], we extend the classic SSA form by using the μ and χ operators [4] to make explicit all potential uses and definitions at loads/stores and call sites, as shown in Figure 2. A load or call site is annotated with a $\mu(v, C_v, P_v)$ operation to indicate a MAY-USE of v in context C_v along path P_v. A store or call site is annotated with a $v = \chi(v, C_v, P_v, M_v)$ operation to indicate a MAY-DEF (MUST-DEF) of v in context C_v along path P_v if M_v is MAY (MUST).

5.3 Capturing Path Correlation

This section focuses on capturing path correlation without strong updates. We consider functions with return statements in Section 5.4. As we do not distinguish MAY-DEFs and MUST-DEFs for now, the last entry $C_{\text{out}}^{\text{must}}$ in a *MOD* function (Definition 5) is ignored and the last entry in a χ operation is always a MAY. For Figure 2, all points-to maps are unchanged except for f_1 and g_1 in *foo*:

$$\begin{array}{l} \mathsf{PtrMap}(\mathrm{f}_1) = (\emptyset, \{(q, x \rightarrow f, \neg P_1), (f, x \rightarrow f, \mathsf{true})\}) \\ \mathsf{PtrMap}(\mathrm{g}_1) = (\{(z, \mathsf{true}, \mathsf{true}), (w, \mathsf{true}, P_1)\}, \emptyset) \end{array} \tag{5}$$

Without path-sensitive strong updates, the old points-to sets (i.e., $\mathsf{PtrMap}(f_0)$ for f_1 and $\mathsf{PtrMap}(g_0)$ for g_1 in Figure 2(b)) must be preserved along path "true" as above and cannot be killed path-sensitively as in Figure 2(b).

To account for the read and modification side effects at a call site, the binding between the actual and formal parameters is performed in the standard manner.

Definition 7 (Mappings of Formal-Ins and Formal-Outs). *For a formal-in $f_{\text{in}} \in \mathbb{F}_{\text{in}}^n$ of method n invoked at call site c, Callee2Caller$_{\text{in}}(c, n, f_{\text{in}})$ denotes the corresponding actual parameter of f_{in} at c if f_{in} is a formal parameter of n and f_{in} itself otherwise (i.e., if f_{in} is a nonlocal). For a formal-out $f_{\text{out}} \in \mathbb{F}_{\text{out}}^m$ of n invoked at c, Callee2Caller$_{\text{out}}(c, n, f_{\text{out}})$ denotes the variable at c that is assigned from f_{out} if f_{out} is a return parameter of n and f_{out} itself otherwise.*

In Figure 2, *Callee2Caller*$_{\text{in}}(c, foo, x) = a$, *Callee2Caller*$_{\text{in}}(c, foo, q) = q$, and *Callee2Caller*$_{\text{in}}(c, foo, f)$=*Callee2Caller*$_{\text{out}}(c, foo, f)$=$f$, where c is the call to *foo*.

Conceptually, SPAS proceeds in the following two sequential stages:

Stage 1. $\mathcal{L} = \textbf{Partition}(\mathbb{O})$ We compute $\mathcal{L}$ by using some fast flow-insensitive pointer analysis for the pointers in $\mathbb{O}$. For example, we can apply Steensgaard's algorithm [18] to obtain a points-to graph, merge all predecessors of each node, and finally, make the points-to graph acyclic by collapsing SCCs, as in [23]. The points-to level of a variable is its longest length over $\{0, \ldots, L\}$ to a sink node.

Stage 2. $\Delta_{-1} = \textbf{Analyze}(\mathcal{L}, \Delta_L, \mathbb{G})$ We build SSA and resolve pointers, level by level, from L to 0. Δ_L is the initial SSA that satisfies vacuously Property 1 for L and $\mathbb{G}$ is the initial call graph constructed when function pointers are not yet resolved. ANALYZE is restarted whenever new points-to relations are discovered for a function pointer. $\mathbb{G}$ is always a directed acyclic graph. In the presence of recursive calls, $\mathbb{G}$ is made acyclic by collapsing all SCCs. The analysis within each SCC is performed iteratively until a fixed-point is reached to obtain full context sensitivity for all the methods in the SCC. Once ANALYZE has run to completion, Δ_{-1} is the full-sparse SSA obtained that satisfies Property 1 for level -1 (excluding its Part(c)) and all pointers have been fully resolved.

When analyzing level ℓ, ANALYZE starts with Δ_ℓ, i.e., the SSA form that satisfies Property 1 for ℓ and ends with producing $\Delta_{\ell-1}$, i.e., the SSA form that satisfies Property 1 for $\ell - 1$. The call graph $\mathbb{G}$ is traversed twice, first bottom-up (reversal topologically) and then top-down (topologically). When points-to cycles are detected, level ℓ is re-reanalyzed until $\Delta_{\ell-1}$ is completely built. Thus, the contexts in a transfer function may comprise the points-to relations of some formal-ins discovered earlier at higher levels and the current level ℓ.

A context used in a callee is mapped to a caller in the standard manner by applying the context mapping introduced in Definition 8 below.

Definition 8 (Context Mapping). *Let C be a context used in a callee n invoked at a call site c in a method m. Callee2Caller$_{\text{ctx}}(c, n, C)$ denotes the mapping of C from callee n to call site c by performing a formal-to-actual parameter mapping. It is understood that every points-to relation in Callee2Caller$_{\text{ctx}}(c, n, C)$ that is not dependent on any of m's contexts is fully evaluated (to* true *or* false*).*

Figure 3 gives our algorithm for analyzing a method m at level ℓ. We describe the bottom-up phase first but both phases may have to be understood together.

To soundly capture path correlation, the path assigned to a variable at any of its definition site must not under-approximate the scope of its definition.

① BOTTOMUP: *Add_μ_χ_Callsites*. Due to Property 1(c), SPAS proceeds to expose the MAY-USEs and MAY-DEFs for each pointer at level ℓ that is accessed at a call site c. This is done by simply examining the context condition $C^{\text{may}}_{f_{\text{out}}}$ of $MOD(n, f_{\text{out}})$ (Definition 5) and the context condition $C_{f_{\text{in}}}$ of $USE(n, f_{\text{in}})$ (Definition 6) of each callee n invoked at c, which were computed earlier during the same bottom-up phase. In line 13, the path for a χ operation is safely over-approximated as P_c, i.e., the path of call site c, where f_{out} may be defined. As SPAS tracks path-sensitivity intraprocedurally, the path condition for a μ operation at a call site is irrelevant and thus marked with a '$*$'.

```
1 BottomUp (Method: m, Level ℓ)
2   Step ①: Add_μ_χ_Callsites(m, ℓ)
3   Step ②: Build_SSA(m, ℓ)
4   Step ③: Pointer_Inference(m, ℓ)
5   Step ④: Comp_MOD_USE_Funs (m, ℓ)

6 ① Add_μ_χ_Callsites(Method: m, Level: ℓ)
7   for each call site c in method m
8     Let P_c be the path allocated to call site c
9     for each callee n invoked at call site c
10      for each formal-out f_out ∈ F^n_out of n,
          L(f_out) = ℓ, that is not a return parameter
11        Let MOD(n, f_out) = (∗, ∗, C^may_{f_out})
12        if (C^may_{f_out} = Callee2Caller_ctx(c, n, C^may_{f_out})) ≠ false
13          Add f_out = χ(f_out, C^may_{f_out}, P_c, MAY) for c
14      for each formal-in f_in ∈ F^n_in of n,
          L(f_in) = ℓ, that is not a formal parameter
15        Let USE(n, f_in) = (∗, C_{f_in})
16        if (C_{f_in} = Callee2Caller_ctx(c, n, C_{f_in})) ≠ false
17          Add μ(f_in, C_{f_in}, ∗) for c

18 ② Build_SSA(Method: m, Level: ℓ)
19   Apply the SSA construction algorithm [5]

20 ③ Pointer_Inference(Method: m, Level: ℓ)
21   Perform a guarded inclusion-based flow-insensitive
     pointer analysis using the rules in Table 1

22 ④ Comp_MOD_USE_Fun(Method: m, Level: ℓ)
23   See text (Section 5.3)

24 TopDown (Method: m, Level ℓ)
25   Step ⑤: Resolve_PointsToSets(m, ℓ)
26   Step ⑥: Add_μ_χ_Derefs(m, ℓ)

27 ⑤ Resolve_PointsToSets(Method: m, Level: ℓ)
28   for each call site c in method m
29     for each callee n invoked at call site c
30       for each variable version p_i in m, L(p) = ℓ,
           such that p_i reaches call site c and p =
           Callee2Caller_in(c, n, f_in), where f_in ∈ F^n_in
31         for each (v, C_v, P_v) ∈ Loc(p_i)
32           PtrSet(f_in) ∪= {v}
33         for each (f, C_f, P_f) ∈ Dep(p_i)
34           PtrSet(f_in) ∪= PtrSet(f)

35 ⑥ Add_μ_χ_Derefs(Method m, Level: ℓ)
36   for each store "∗p_i = ..." in m, L(p) = ℓ
37     Let PtrMap(p_i) = (Loc(p_i), Dep(p_i))
38     for each (v, C_v, P_v) ∈ Loc(p_i)
39       Add v = χ(v, C_v, P_v, MAY)
40     for each (f_in, C_{f_in}, P_{f_in}) ∈ Dep(p_i)
41       for each v ∈ PtrSet(f_in)
42         Add v = χ(v, C_{f_in} ∧ f_in → v, P_{f_in}, MAY)
43   for each load "· · · = ∗p_i" in m, L(p) = ℓ
44     Let PtrMap(p_i) = (Loc(p_i), Dep(p_i))
45     for each (v, C_v, P_v) ∈ Loc(p_i)
46       Add μ(v, C_v, P_v)
47     for each (f_in, C_{f_in}, P_{f_in}) ∈ Dep(p_i)
48       for each v ∈ PtrSet(f_in)
49         Add μ(v, C_{f_in} ∧ f_in → v, P_{f_in}, MAY)
```

Fig. 3. Bottom-up and top-down analysis of method m at level ℓ

Let us see how the MAY-USE and MAY-DEF are added for the call site c_{foo} to *foo* in *main* in Figure 2(b), given the transfer functions of *foo* in (4). In line 12, $\overline{C^{\text{may}}_f} = \mathit{Callee2Caller}_{\text{ctx}}(c_{foo}, foo, C^{\text{may}}_f) = \mathsf{true}$ since by Definition 8, $C^{\text{may}}_f = x \rightarrow f$ is mapped to $a \rightarrow f$ at the call site, which is generated locally in *main*. So the MAY-DEF, $f = \chi(f, \mathsf{true}, \mathsf{true}, \text{MAY})$, is added. The MAY-USE, $\mu(q, \mathsf{true}, *)$, is added since $C_q = \mathsf{true}$ in $\mathit{USE}(foo, q)$.

② BottomUp: ***Build_SSA.*** Once all MAY-USEs and MAY-DEFs are exposed for the pointers at level ℓ accessed, they can be put in SSA by applying the classic SSA construction algorithm [5], as illustrated in Figure 2.

③ BottomUp: ***Pointer_Inference.*** Table 1 lists the seven rules for resolving the points-to maps for the pointers at level ℓ in a method m. The first six rules are illustrated in Figure 2 and the last partially when *Add_μ_χ_Callsites* is discussed.

The propagation of points-to information across an assignment may be guarded by both a context condition and a path condition. We define $\mathcal{P}(x) \times C_x \times P_x = \{(v, C_v \wedge C_x, P_v \wedge P_x) \mid (v, C_v, P_v) \in \mathcal{P}(x)\}$. $\mathcal{P}(x) \cup \mathcal{P}(y)$ includes all and only elements in $\mathcal{P}(x)$ and $\mathcal{P}(y)$ such that if $(v, C^x_v, P^x_v) \in \mathcal{P}(x)$ and $(v, C^y_v, P^y_v) \in \mathcal{P}(y)$, then both are merged as $(v, C^x_v \vee C^y_v, P^x_v \vee P^y_v)$.

Loc-Init is self-explanatory. As SPAS is intraprocedurally path-sensitive, the path $P_{p_i = \&a}$ is generated locally in method m. *Dep-Init* is applied to a copy of the form $p_0 = p$, where p is a formal-in of method m. Such copies are added at

Table 1. Rules for resolving points-to maps $\mathsf{PtrMap}(x) = \{\mathsf{Loc}(x), \mathsf{Dep}(x)\}$ in method m for level ℓ. Each of the last five is applied once for $\mathbb{P} = \mathsf{Loc}$ and once for $\mathbb{P} = \mathsf{Dep}$.

Rule	Statement	Constraints	Inference Operations
Loc-Init	$p_i = \&a$ (on path $P_{p_i=\&a}$)	$p_i \supseteq \{a\}$	$\mathsf{Loc}(p_i) = \{(a, \mathsf{true}, P_{p_i=\&a})\}$ $\mathsf{Dep}(p_i) = \emptyset$
Dep-Init	$p_0 = p$ (a formal-in)	$p_0 \supseteq p$	$\mathsf{Loc}(p_0) = \emptyset$ $\mathsf{Dep}(p_0) = \{(p, \mathsf{true}, \mathsf{true})\}$
Assn	$p_i = q_j$ (on path $P_{p_i=q_j}$)	$p_i \supseteq_{\mathsf{true}\times P_{p_i=q_j}} q_j$	$\mathcal{P}(p_i) = \mathcal{P}(q_j) \times \mathsf{true} \times P_{p_i=q_j}$
Mu	$\mu(v_k, C_{v_k}, P_{v_k})$ $p_i = *q_j$	$p_i \supseteq_{C_{v_k}\times P_{v_k}} v_k$	$\mathcal{P}(p_i) \cup\!= \mathcal{P}(v_k) \times C_{v_k} \times P_{v_k}$
Chi	$*p_i = q_j$ $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \mathrm{MAY})$	$v_s \supseteq_{C_{v_s}\times P_{v_s}} q_j$ $v_s \supseteq v_t$	$\mathcal{P}(v_s) \cup\!= \mathcal{P}(q_j) \times C_{v_s} \times P_{v_s}$ $\mathcal{P}(v_s) \cup\!= \mathcal{P}(v_t)$
Phi	$p_i{=}\phi(p_j, p_k)$ ($P^{\phi}_{p_j}$ ($P^{\phi}_{p_k}$) is the path of the incoming edge along which the value of p_j (p_k) flows into p_i)	$p_i \supseteq_{\mathsf{true}\times P^{\phi}_{p_j}} p_j$ $p_i \supseteq_{\mathsf{true}\times P^{\phi}_{p_k}} p_k$	$\mathcal{P}(p_i) = \mathcal{P}(p_j) \times \mathsf{true} \times P^{\phi}_{p_j}$ $\mathcal{P}(p_i) = \mathcal{P}(p_k) \times \mathsf{true} \times P^{\phi}_{p_k}$
Call	call site c invoking callee n $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \mathrm{MAY})$ (P_{v_s} is the path P_c of c inserted in line 13 in Figure 3)	C1 $v = \mathit{Callee2Caller}_{\mathrm{out}}(c, n, f_{\mathrm{out}})$ C2 Let $\mathit{MOD}(n, f_{\mathrm{out}}) = (\mathsf{Loc}(f_{\mathrm{out}}), \mathsf{Dep}(f_{\mathrm{out}}), *)$ C3 **for** every $(o, C_o, *) \in \mathsf{Loc}(f_{\mathrm{out}})$ C4 **Generate** $v_s \supseteq_{\mathit{Callee2Caller}_{\mathrm{ctx}}(c,n,C_o)\times P_{v_s}} \{(o, \mathsf{true}, \mathsf{true})\}$ C5 **for** every $(f_{\mathrm{in}}, C_{f_{\mathrm{in}}}, *) \in \mathsf{Dep}(f_{\mathrm{out}})$ C6 $w = \mathit{Callee2Caller}_{\mathrm{in}}(c, n, f_{\mathrm{in}})$ such that w_i reaching c C7 **Generate** $v_s \supseteq_{\mathit{Callee2Caller}_{\mathrm{ctx}}(c,n,C_{f_{\mathrm{in}}})\times P_{v_s}} w_i$ C8 **Generate** $v_s \supseteq v_t$	

the entry of m for all its formal-ins. The path condition is over-approximated as true since p_0 may point to whatever p point to on entry of the method considered.

Assn applies to every other copy assignment. The points-to relations at the RHS are propagated to the LHS, guarded by the path of the assignment.

Rules *Mu* and *Chi* are also easy to understand. The context and path conditions in a χ or μ operation serve as the guards to enforce context- and path-sensitivity. According to the second constraint for *Chi*, the old points-to relations of v are weakly updated, i.e., simply propagated from v_t to v_s unchanged.

Let us consider Rule *Phi*. For each operand, we use the path along which its value flows into the result as the guard to propagate its points-to relations into the result. In a FSCS pointer analyzer that does not consider path-sensitivity, the two unguarded constraints $p_i \supseteq p_j$ and $p_i \supseteq p_k$ are generated. Applying these two would yield the two spurious points-to relations shown in Figure 1(d).

Finally, Rule *Call* is applied to a call site c in the standard manner. In lines C4 and C7, constraints are generated to propagate the points-to relations created by a callee n in $\mathsf{Loc}(f_{\mathrm{out}})$ and $\mathsf{Dep}(f_{\mathrm{out}})$ to v_s, guarded by the (mapped) context conditions C_o and $C_{f_{\mathrm{in}}}$, respectively; but the paths created inside the callee are ignored (and hence, the $*$'s). In line C8, v is weakly updated as in Rule *Chi*.

④ BottomUp: ***Comp_MOD_USE_Funs.*** For each formal-out $f_{\mathrm{out}} \in \mathbb{F}^m_{\mathrm{out}}$ at level ℓ, we write $f^{\max}_{\mathrm{out}}$ for its last SSA version in method m. Let $\mathsf{PtrMap}(f^{\max}_{\mathrm{out}}) = (\mathsf{Loc}(f^{\max}_{\mathrm{out}}), \mathsf{Dep}(f^{\max}_{\mathrm{out}}))$, which is already available. Then $\mathit{MOD}(m, f_{\mathrm{out}})$ is defined to be $(\mathsf{Loc}(f^{\max}_{\mathrm{out}}), \mathsf{Dep}(f^{\max}_{\mathrm{out}}), C^{\mathrm{may}}_{f_{\mathrm{out}}})$, where $C^{\mathrm{may}}_{f_{\mathrm{out}}}$ is set as true if f_{out} is directly modified in method m and set otherwise as a disjunction of the context conditions in all its MAY-DEF sites, i.e., all χ operations of $f^{\max}_{\mathrm{out}}$ in method m.

For a formal-in $f_{\mathrm{in}} \in \mathbb{F}^m_{\mathrm{in}}$ at level ℓ, we write f^0_{in} for its first SSA version in m. Thus, $\mathit{USE}(m, f_{\mathrm{in}}) = (\mathsf{PtrSet}(f_{\mathrm{in}}), C_{f_{\mathrm{in}}})$, where $C_{f_{\mathrm{in}}}$ is true if f^0_{in} is directly used

in method m and otherwise as a disjunction of the context conditions at all MAY-USE sites, i.e., μ operations of $f^0_{f_{\text{in}}}$ in m. $\mathsf{PtrSet}(f_{\text{in}})$ is computed later by *Resolve_PointsToSets* and subsequently used in lines 41 and 48 of *Add_μ_χ_Derefs*.

In Figure 2(b), the *MOD* and *USE* functions of *foo* are given in (4).

⑤ TopDown: ***Resolve_PointsToSets.*** The points-to sets of the pointers at level ℓ in method m can now be obtained by resolving all formal-ins (lines 32 and 34).

⑥ TopDown: *Add_μ_χ_Derefs.* We annotate all dereferences to the pointers at level ℓ with MAY-USEs and MAY-DEFs. The points-to relations in $\mathsf{Loc}(p_i)$ are generated locally in method m and handled straightforwardly. To deal with those generated by m's callers in $\mathsf{Dep}(p_i)$ in lines 42 and 49, new context conditions are generated. If p_i points to v, because a formal-in f_{in} does, then $C_{f_{\text{in}}}$ is strengthened to include $f_{\text{in}} \rightarrow v$ to indicate the context condition under which the MAY-USE/MAY-DEF occurs (Figure 2).

SPAS soundly tracks path correlation on top of a FSCS pointer analyser.

Theorem 1. *PtrSet*(p) *contains all possible targets for* p *during any execution.*

Proof sketch. In a FSCS pointer analyser without considering path-sensitivity, the path/scope for a variable definition is taken as true. SPAS refines but never under-approximates it at a call site (lines 8 and 13 in Figure 3) and in Rules *Loc-Init*, *Dep-Init*, *Assn* and *Phi* (Table 1). So the soundness of SPAS follows from that of the underlying FSCS analyser, which preserves Property 1 level-wise.

The following theorem states a well-known fact about path-sensitivity.

Theorem 2. *Let* $\mathit{PtrSet}^{\mathit{SPAS}}(p)$ *(*$\mathit{PtrSet}^{\mathit{LevPA}}(p)$*) be the points-to set of* p *found by SPAS (a FSCS pointer analyser like LevPA). Then* $\mathit{PtrSet}^{\mathit{SPAS}}(p) \subseteq \mathit{PtrSet}^{\mathit{LevPA}}(p)$.

Proof sketch. Compared to LevPA, as argued in the proof of Theorem 1, the path condition at a variable definition site in SPAS is either the same or strengthened.

5.4 Supporting Strong Updates

In Table 1, a χ operation $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \text{MAY})$ represents a MAY-DEF, where P_{v_s} safely overapproximates the scope where v_s is defined (Theorem 1). In Rules *Chi* and *Call*, $v_s \supseteq v_t$ is always used as only weak updates are allowed.

To support strong updates, we consider a χ operation, $v_s = \chi(v_t, C_{v_s}, P_{v_s}, M_{v_s})$, associated with a store "$*p_i = \ldots$" residing on a path P_χ in method m. Let this χ operation be referred to as χ_{op}. In χ_{op}, $M_{v_s} \in \{\text{MAY}, \text{MAY}^+, \text{MUST}\}$. So MAY$^+$ is now identified as a special case of MAY introduced in Section 5.2. M_{v_s} is set as MUST when χ_{op} is a MUST-DEF. This is both context-sensitive and path-sensitive, meaning that p_i must point to v along path P_{v_s} in context C_{v_s}. However, in the other contexts, $p_i \rightarrow v$ may not hold, i.e., P_{v_s} may not be exact. M_{v_s} is MUST when $\mathsf{UniqueTarget}(m, p_i, v)$ is true, which is defined to hold when (a) p_i points to v uniquely whenever method m is invoked at a call site such that p_i points to v and (b) v is a concrete object in Singletons. Following [14], Singletons is the subset of locations in $\mathbb{O}$ with arrays, heap objects and locals inside recursion

cycles being removed. M_{v_s} is set as MAY$^+$ or MAY when χ_{op} is a MAY-DEF, in which case, dereferencing p_i may yield more than one target. M_{v_s} is set as MAY$^+$ when $C_{v_s} = \mathsf{true}$ and P_{v_s} is exact, meaning that p_i must point to v whenever the program is executed along P_{v_s}. Otherwise, M_{v_s} is set as MAY, in which case, p_i may or may not point to v as described in Section 5.3.

Now, constraint $v_s \supseteq v_t$ used in Table 1 is augmented with the guards:

$$v_s \supseteq_{C_{\text{old}} \times P_{\text{old}}} v_t, \quad \text{where}(C_{\text{old}}, P_{\text{old}}) = \begin{cases} (\neg C_{v_s}, \mathsf{true}) \text{ if } M_{v_s} = \text{MUST} \\ (\mathsf{true}, P_\chi \wedge \neg P_{v_s}) \text{ if } M_{v_s} = \text{MAY}^+ \\ (\mathsf{true}, \mathsf{true}) \text{ if } M_{v_s} = \text{MAY} \end{cases} \quad (6)$$

The base version of our algorithm, shown in Figure 3, performs weak updates only as it treats MUST and MAY$^+$ conservatively as MAY. In our fully-fledged algorithm, SPAS obtains improved precision since *all-path strong updates*, much like Dead Code Elimination (DCE), are enabled when M_{v_s} is MUST. In this case, the old points-to relations of v_t at all incoming paths of the store $p_i = \ldots$ are killed *if they are in the same context* C_{v_s}. In addition, SPAS improves analysis precision further since *some-path strong updates*, must like Partial DCE [22], are also performed when M_{v_s} is MAY$^+$. In this case, the old points-to relations of v_t *in any context* are killed along P_{v_s} but allowed to flow into v_s along $P_\chi \wedge \neg P_{v_s}$.

We only need to make small changes to our algorithm in Figure 3. Only the path conditions for the points-to relations of p_i established intraprocedurally in $\mathsf{Loc}(p_i)$ may be considered as being exact conservatively.

Line 8. Insert a MUST-DEF, $r = \chi(r, \mathsf{true}, P_c, \text{MUST})$, between lines 8 and 9 to handle the assignment of a return parameter in a function invoked at the call site c to a locally-defined pointer r in method m (Definition 2).

Line 11. Use $MOD(m, f_{\text{out}})$ given in Definition 5, where its fourth component $C^{\text{must}}_{\text{out}}$ is a conjunction of context conditions, one condition C_P for every possible path P from the entry to the exit of method m, such that C_P is true if f_{out} is directly modified on P and a disjunction of context conditions in all χ's representing MUST-DEFs on P otherwise.

Line 13. M_{v_s} is MUST if $Callee2Caller_{\text{ctx}}(c,n,C^{\text{must}}_{\text{out}})$ holds for every callee n checked in line 9 and MAY otherwise.

Line 39. M_{v_s} is MAY$^+$ if $v \in \mathsf{Singletons}$ and p_i is not defined in a cycle in the CFG of method m (decided in *Pointer_Inference*), and MAY otherwise.

Line 42. M_{v_s} is MUST if $\mathsf{UniqueTarget}(m, p_i, v)$ holds and MAY otherwise. (This overwrites MAY$^+$ set for v in line 39 if $\mathsf{UniqueTarget}(m, p_i, v)$ holds.)

The points-to maps for f_1 and g_1 are thus refined from (5) to those in Figure 2.

Theorem 3. *With strong updates thus specified, Theorems 1 and 2 remain valid.*

Proof sketch. Follows simply from the definitions of MUST, MAY$^+$ and MAY.

6 Experimental Evaluation

We have implemented SPAS in the Open64 compiler (v4.2). We use the CUDD2.4.2 library for representing points-to relations, contexts and paths. As in

Table 2. Percentage increases of analysis overhead under SPAS w.r.t. LevPA

Benchmark	Analysis Overhead				SPAS								
	Time (secs)		Memory (MBs)		D-Vars	Time Breakdown (secs)							
	LevPA	SPAS(%)	LevPA	SPAS(%)		Comp. Levels	Gen. Paths	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
164.gzip	0.42	9.52	20.97	9.31	269	0.09	0.02	0.07	0.22	0.02	0.00	0.04	0.00
175.vpr	1.11	12.00	55.78	9.62	639	0.34	0.03	0.09	0.50	0.15	0.01	0.08	0.04
176.gcc	1230.76	23.62	6576.05	9.91	16043	4.05	1.02	129.99	346.20	926.12	15.70	95.73	2.71
177.mesa	8.21	19.01	247.29	12.41	6242	2.84	0.25	0.48	2.90	0.83	0.11	0.23	2.13
179.art	0.08	0.00	5.28	10.51	47	0.03	0.00	0.00	0.03	0.01	0.00	0.00	0.01
181.mcf	0.13	0.00	5.74	7.52	34	0.03	0.00	0.00	0.05	0.01	0.00	0.01	0.03
183.equake	0.09	22.20	5.62	10.47	50	0.04	0.01	0.00	0.05	0.00	0.01	0.00	0.00
186.crafty	3.65	62.73	136.44	11.33	1517	0.55	0.06	0.48	3.06	1.55	0.11	0.07	0.06
188.ammp	2.28	35.53	58.94	5.93	804	0.03	0.09	1.11	1.54	0.03	0.06	0.20	0.03
197.parser	15.31	13.46	133.60	10.60	570	0.27	0.03	0.23	1.99	13.44	0.04	1.20	0.17
254.gap	21.71	12.16	440.50	4.90	7482	1.91	0.31	4.92	7.43	4.20	0.51	4.23	0.84
255.vortex	19.37	27.36	624.01	5.24	6019	1.91	0.33	4.88	8.69	3.67	0.44	4.30	0.45
256.bzip2	0.20	0.00	13.29	10.45	144	0.06	0.00	0.03	0.07	0.01	0.02	0.01	0.00
300.twolf	1.65	21.82	64.22	7.94	520	0.52	0.03	0.09	0.80	0.37	0.00	0.08	0.12
400.perlbench	971.20	24.75	4111.17	9.11	13218	2.98	0.87	105.84	277.65	680.99	13.01	125.60	4.67
401.bzip2	0.79	36.71	24.52	16.68	530	0.17	0.02	0.03	0.66	0.07	0.01	0.01	0.11
429.mcf	0.11	18.18	4.95	24.45	37	0.03	0.00	0.00	0.03	0.03	0.00	0.00	0.04
433.milc	0.87	0.00	45.05	10.23	469	0.32	0.02	0.17	0.19	0.08	0.01	0.04	0.04
445.gobmk	14.66	12.21	682.00	16.64	3680	1.45	0.23	3.14	5.83	2.95	0.22	2.26	0.37
456.hmmer	2.71	18.15	45.97	14.00	1673	0.86	0.05	0.12	1.30	0.50	0.03	0.10	0.86
458.sjeng	1.39	21.58	55.78	11.93	1060	0.27	0.06	0.24	0.65	0.34	0.01	0.10	0.02
462.libquantum	0.32	12.50	0.00	10.41	141	0.07	0.02	0.10	0.11	0.03	0.00	0.02	0.01
464.h264ref	5.77	39.86	247.29	11.06	2457	1.44	0.16	0.80	2.97	1.37	0.16	0.47	0.70
470.lbm	0.07	14.29	5.28	11.52	19	0.04	0.00	0.01	0.02	0.00	0.00	0.00	0.01
482.sphinx3	1.76	2.84	5.74	11.98	835	0.53	0.07	0.13	0.65	0.16	0.02	0.08	0.17

[23], parameterised spaces are used to reduce analysis overhead and improve precision. We evaluate the scalability of SPAS in handling (intraprocedural) path-sensitivity by integrating it with a state-of-the-art FSCS pointer analyser, LevPA [23], which already performs all-path strong updates (for MUST-DEFs). Despite this, SPAS obtains points-to information with non-decreasing precision with improvements at stores/loads that are amenable to path-sensitive pointer analysis, at a small increase in analysis overhead. We have used all 27 C benchmarks from SPEC CPU2000 and CPU2006 and carried out our experiments on a 3.0GHz quad-core Intel Xeon system running Redhat Enterprise Linux 5 (kernel version is 2.6.18) with 16GB memory. Benchmarks `253.perlbmk` and `403.gcc` run out of memory under both analyzers and are thus excluded in further discussions.

6.1 Analysis Overhead

As shown in Table 2, SPAS uses 18.42% more time and 10.97% more memory than LevPA on average. To the best of our knowledge, SPAS is the fastest path-sensitive pointer analysis reported. Benchmarks `176.gcc` and `400.perlbench` are the most costly to analyze due to many iterations required for handling function pointers and recursion cycles. From the statistics in the last nine columns, we can see the extra analysis overhead incurred by SPAS. Column "D-Vars" gives the number of decision variables used to encode the paths in a program using BDDs, resulting in a slight increase in the total memory usage by SPAS for each benchmark (measured using the memory tracing tool available in Open64).

In the last eight columns, the analysis time for a benchmark is broken down into eight parts on computing points-to levels, generating paths (Section 5.1),

Table 3. Percentages of variables at χ's and μ's with more accurate points-to sets

Benchmark	χ's (MAY/MUST-DEFs at Stores)					μ's (MAY-USEs at Loads)				
	LevPA (Total)	SPAS (More Precise)				LevPA (Total)	SPAS (More Precise)			
		Traditional		Path-Sensitive			Traditional		Path-Sensitive	
		Total	%	Total	%		Total	%	Total	%
164.gzip	144	6	4.17	115	79.86	165	4	2.42	128	77.58
175.vpr	232	6	2.59	135	58.19	184	7	3.80	106	57.61
176.gcc	3710	132	3.56	1356	36.55	11843	506	4.27	4701	39.69
177.mesa	3780	16	0.42	1101	29.13	5200	60	1.15	1375	26.44
179.art	6	0	0.00	6	100.00	6	0	0.00	6	100.00
181.mcf	76	0	0.00	26	34.21	144	0	0.00	65	45.14
183.equake	29	0	0.00	2	6.90	55	0	0.00	5	9.09
186.crafty	343	23	6.71	291	84.84	1007	62	6.16	860	85.40
188.ammp	475	20	4.21	418	88.00	694	46	6.63	615	88.62
197.parser	374	14	3.74	296	79.14	403	19	4.71	260	64.52
254.gap	297	6	2.02	188	63.30	5466	32	0.59	3311	60.57
255.vortex	801	11	1.37	120	14.98	3651	36	0.99	469	12.85
256.bzip2	51	0	0.00	13	25.49	109	0	0.00	33	30.28
300.twolf	106	3	2.83	33	31.13	265	17	6.42	111	41.89
400.perlbench	2938	146	4.97	1045	35.57	17084	641	3.75	7027	41.13
401.bzip2	601	13	2.16	96	15.97	1380	11	0.80	184	13.33
429.mcf	77	3	3.90	29	37.66	162	2	1.23	45	27.78
433.milc	153	0	0.00	29	18.95	287	0	0.00	34	11.85
445.gobmk	1123	58	5.16	382	34.02	2957	139	4.70	1795	60.70
456.hmmer	568	37	6.51	305	53.70	1680	119	7.08	1239	73.75
458.sjeng	343	13	3.79	268	78.13	748	32	4.28	706	94.39
462.libquantum	5	0	0.00	0	0.00	7	0	0.00	0	0.00
464.h264ref	1994	68	3.41	477	23.92	7654	121	1.58	1522	19.89
470.lbm	8	0	0.00	0	0.00	13	0	0.00	0	0.00
482.sphinx3	220	8	3.64	66	30.00	782	13	1.66	208	26.60

and performing the six steps of SPAS in Figure 3. In 13 out of 25 benchmarks, Step 2 (*Build_SSA*) and Step 3 (*Pointer_Inference*) consume most of the analysis time in a benchmark. These are also the very steps where SPAS spends more analysis time than LevPA as it does extra work in handling program paths. For `186.crafty`, `188.ammp`, `401.bzip2` and `464.h264ref`, the analysis times under LevPA are small. SPAS adds relatively high overheads mainly in Steps 2 and 3.

6.2 Path-Sensitive Precision

Table 3 shows that SPAS can obtain more precise points-to sets than LevPA at certain loads/stores in most benchmarks. We consider only the loads/stores that reside beyond the first branch in the CFG of a method after all its SCCs (strongly connected components) have been collapsed. The pointers accessed indirectly at their associated χ and μ operations (MAY/MUST-DEFs and MAY-USEs) are the ones whose points-to information may be improved by SPAS.

We measure the number of χ's and μ's with improved points-to information in two ways, indicated by their *Traditional* and *Path-Sensitive* columns. Note that by Theorem 2, $\mathsf{PtrSet}(p)^{\mathsf{SPAS}} \subseteq \mathsf{PtrSet}(p)^{\mathsf{LevPA}}$ holds for any pointer p. With the traditional metric, the points-to set of p is said to be *more precise* under SPAS if $|\mathsf{PtrSet}(p)^{\mathsf{SPAS}}| < |\mathsf{PtrSet}(p)^{\mathsf{LevPA}}|$. With the path-sensitive metric, the path, i.e., the scope information governing each points-to target is also taken into account. For LevPA, the path guarding a χ or μ operation is always true. Thus, the points-to set of p is *more precise* under SPAS if either $|\mathsf{PtrSet}(p)^{\mathsf{SPAS}}| < |\mathsf{PtrSet}(p)^{\mathsf{LevPA}}|$ or the guarding path for a χ or μ operation is not true (i.e., more restricted).

As shown in Table 3, SPAS has improved points-to information in most benchmarks. Under "Traditional" for χ's, the percentage improvements range from 0% to 6.71% with an average of 2.61%. Under "Traditional" for μ's, the percentages are within 0 to 7.08% with an average of 2.49%. Under "Path-Sensitive", the improvements are more significant with an average of 42.38% for χ's and 41.07% for μ's. These results should be understood with some caveats. First, what SPAS is compared with is a state-of-the-art FSCS pointer analyser that already performs all-path strong updates. Second, SPAS obtains such improved path-sensitive precision at small analysis overhead. Finally, such improvement can be critical for some client applications (e.g., bug detection).

Let us look at some benchmarks in detail. In the case of `164.gzip`, `176.gcc` `197.parser`, `400.perlbench`, `429.mcf`,`464.h264ref`, `482.sphinx3`, `458.sjeng` the improvements are mostly over 3% for both χ's and μ's. Path-sensitive analysis provides little benefits for seven benchmarks: `179.art`,`181.mcf`,`183.equake`, `256.bzip2`, `433.milc`, `462.libquantum` and `470.lbm`. They are small programs with few pointers but mostly scalar-to-array assignments. For other benchmarks on scientific computations, such as `177.mesa`, and `255.vortex` , the improvements are below 2%. In contrast, benchmarks such as `186.crafty`, `445.gobmk` and `456.hmmer` exhibit much better precision improvements. They each have a relatively high number of decision variables, giving rise to more opportunities for capturing path correlation.

7 Conclusion

We have presented SPAS, a path-sensitive pointer analysis that extends a recent flow- and context-sensitive pointer analysis LevPA. Our experimental evaluation shows that SPAS incurs reasonable analysis overhead over LevPA (on average 18.42% increase in analysis time and 10.97% increase in memory usage) and computes more precise points-to information. Our results are expected to provide insights for developing client-driven pointer analysis techniques.

Acknowledgement. This work is supported by the Australian Research Council Grant DP0987236.

References

1. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: PLDI 2003, vol. 38(5), p. 114 (2003)
2. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. In: SPE 2000, vol. 30, pp. 775–802 (2000)
3. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL 1999, p. 146. ACM (1999)
4. Chow, F., Chan, S., Liu, S., Lo, R., Streich, M.: Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 253–267. Springer, Heidelberg (1996)

5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 490 (1991)
6. Dillig, I., Dillig, T., Aiken, A.: Sound, complete and scalable path-sensitive analysis. In: PLDI 2008, pp. 270–280. ACM (2008)
7. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI 1994, pp. 242–256 (1994)
8. Gutzmann, T., Lundberg, J., Lowe, W.: Towards path-sensitive points-to analysis. In: SCAM 2007, pp. 59–68. IEEE (2007)
9. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: POPL 2009, pp. 226–238. ACM (2009)
10. Hardekopf, B., Lin, C.: Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In: CGO 2011 (2011)
11. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: PLDI 2008, pp. 249–259. ACM (2008)
12. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. In: PLDI 1992, vol. 27(7), pp. 235–248 (1992)
13. Le, W., Soffa, M.L.: Refining buffer overflow detection via demand-driven path-sensitive analysis. In: PASTE 2007, pp. 63–68. ACM (2007)
14. Lhoták, O., Andrew Chung, K.-C.: Points-to analysis with efficient strong updates. In: POPL 2011, pp. 3–16. ACM, New York (2011)
15. Benjamin Livshits, V., Lam, M.S.: Tracking pointers with path and context sensitivity for bug detection in c programs. In: FSE 2003, pp. 317–326 (2003)
16. Nystrom, E.M., Kim, H.S., Hwu, W.W.: Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 165–180. Springer, Heidelberg (2004)
17. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: POPL 2011, pp. 17–30. ACM (2011)
18. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL 1996, pp. 32–41. ACM, New York (1996)
19. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI 2004, pp. 131–144. ACM (2004)
20. Xie, Y., Aiken, A.: Context-and path-sensitive memory leak detection. In: FSE 2005, vol. 30(5), p. 125 (2005)
21. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: POPL 2005, vol. 40(1), pp. 351–363 (2005)
22. Xue, J., Cai, Q., Gao, L.: Partial dead code elimination on predicated code regions. In: SPE 2006, vol. 36, pp. 1655–1685 (2006)
23. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In: CGO 2010, pp. 218–229. ACM, New York (2010)
24. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: PLDI 2004, p. 157. ACM (2004)

On the Strength of Owicki-Gries for Resources

Alexander Malkis and Laurent Mauborgne

IMDEA Software Institute

Abstract. In multithreaded programs data are often separated into lock-protected resources. Properties of those resources are typically verified by modular, Owicki-Gries-like methods. The modularity of the Owicki-Gries method has its price: proving some properties may require manual introduction of auxiliary variables. What properties can be proven without the burden of introducing auxiliary variables? We answer this question in the abstract interpretation framework. On one hand, we reveal a lattice structure of the method and supply a syntax-based abstract transformer that describes the method *exactly*. On the other hand, we bound the loss of precision from *above* and *below* by transition-relation-independent weakly relational closures. On infinitely many programs the closures coincide and describe the precision loss exactly; in general, the bounds are strict. We prove the absence of a general exact closure-based fixpoint characterization of the accuracy of the Owicki-Gries method, both in the collecting semantics and in certain trace semantics.

1 Introduction

The paper will characterize the accuracy of a popular verification method for proving safety properties of concurrent programs that operate on resources.

A program operating on resources is a multithreaded program in which threads communicate via sequentially consistent shared memory and in which all shared variables are partitioned into disjoint resources. Each resource may be *available* or *busy*. To access a variable belonging to a resource, a thread waits until the resource gets available and then starts a critical section for that resource, thus making the resource busy. While staying in the critical section, the thread can read and write the resource data, and no other thread can enter a critical section for the same resource, so no other thread can access the resource data. After the thread finishes accessing the resource, it exits the corresponding critical section, making the resource available.

Simple safety properties of such programs can be proven modularly. A modular proof of a program consists of an annotation per control flow location and an annotation per resource. Roughly, an annotation of a control flow location describes the valuations of the variables that the thread may access at that location. An annotation of a resource describes, roughly, the state of the resource when it is available. The annotations of locations have to be sequentially consistent similarly to the standard Hoare-style assertional logic, but with two changes: when a thread acquires a resource, the proof may assume that the invariant of the acquired resource holds, and on releasing a resource the resource invariant should be reestablished.

Such modularity incurs a loss of precision: for many programs, too strong but valid properties cannot be proven by the method. To prove such properties, the program can

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 172–187, 2011.

be manually augmented with so-called auxiliary variables. A set of variables is called auxiliary if, intuitively, projecting the traces of the program to the other variables gives the same result as removing all the statements involving the auxiliary variables and projecting afterwards. A modular proof is created for the augmented program, then the proven property is projected onto the original variables. The ability to use auxiliary variables makes the proof system complete. So far, auxiliary variables have been introduced purely manually, while the construction of the remaining proof can be automated.

We will estimate the loss of precision inherent to the *core* of the Owicki-Gries method, which consists of all the proof rules of the Owicki-Gries logic except the ability to use auxiliary variables, in abstract interpretation.

We will show a rich lattice structure of the Owicki-Gries-core proofs and a syntax-based abstract transformer describing the Owicki-Gries core *exactly*.

We will observe that there is no transition-relation-independent approximation operator that exactly characterizes the loss of precision of the Owicki-Gries core in the collecting semantics. We will also note the absence of equivalent trace-based characterizations of certain kinds.

We will find an approximation operator that induces a useful upper bound on this set, i.e., that will describe how much precision the Owicki-Gries core always loses. This approximation operator depends only on the syntactic structure of the program, but not on its exact transition relation. If a property is provable by the Owicki-Gries core, it is provable by abstract interpretation with this approximation.

We will present an approximation operator inducing a lower bound on the set, i.e., describing how much precision the Owicki-Gries core always retains. This approximation operator also depends only on the syntactic structure of the program, but not on its exact transition relation. If a property is provable by abstract interpretation with this approximation, it is provable by the Owicki-Gries core.

In passing, we will demonstrate an infinite class of simple programs for which both bounds are equal and a program on which both bounds are strict.

In short, the main results and their position are depicted in Fig. 1.

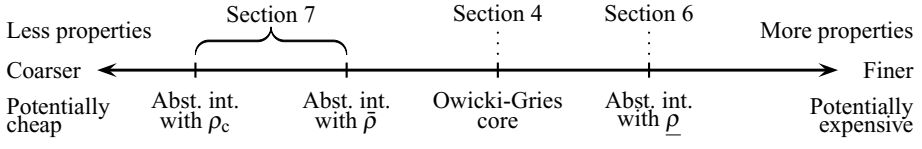


Fig. 1. Precision of analyses. The upper bound is $\underline{\rho}$, the lower bound is $\bar{\rho}$. In addition to $\bar{\rho}$, we will consider a simpler lower bound ρ_c, which is coarser than $\bar{\rho}$, but easier to understand.

The rigorous formalization, computations, proofs, recommendations for practitioners, and comparison to abstract interpretations for general programs are found in the appendix of the technical report [12] appendix of the technical report [12].

2 Resource-Manipulating Language and Its Semantics

2.1 Language RPL

Now we briefly recall the parallel language RPL (Restricted Parallel Language, rigorously defined in [15]) in which shared data are separated into resources and access to a resource is granted exclusively.

Let threads be indexed by the elements of the set *Tid* and, without loss of generality, start their executions together. Threads operate on data variables from the set *DataVar*. A *resource* is a set of data variables; the resources are disjoint. Let *Res* be the set of all resources.

A *statement* is either

- an atomic statement, i.e., an assignment or an "`assume` ϕ" (which skips if the argument expression evaluates to *true* and blocks otherwise),
- a sequential composition of two statements, an `if-then-else`, or a `while` loop,
- a critical section `with` r `when` ϕ `do` C `endwith` where r is a resource, ϕ a formula over data variables and C some statement. We write `with` r `do` C `endwith` when ϕ is `true`.

A thread executes a statement.

A data variable is called *local* to thread t if all the assignments to the variable are syntactically in t. If a variable appears in thread t, it should belong to a resource or be local to t. If a variable belongs to a resource r, it can appear only inside a critical section for r, i.e., inside a "`with` r`...`" statement.

Each resource r is associated with a set of *proof variables* $ProofVar(r)$, which are all data variables that are not assigned except maybe in critical sections for r. All our examples will satisfy $ProofVar(r) = r$ (in general, $ProofVar(r) \supseteq r$).

An RPL program is described by a set of threads, a set of resources and an initial condition on the variables.

2.2 Semantics of RPL

To connect to abstract interpretation, we describe programs in RPL as transition systems.

For each thread t we introduce a fresh non-data program counter variable pc_t, which

- is local to thread t, but not to any other thread, and
- doesn't belong to any resource, and
- takes values from the set of control flow locations *PL* (Program Location).

Let *PCVar* be the set of all program counter variables and $Var = DataVar \cup PCVar$.

We associate each control flow point $p \in PL$ with a set $rsc(p)$ of resources r such that p is inside a `with` r`...` statement.

We allow the same control flow location, say, p, to occur in different threads.

A state of a program is a map from *Var* to the set of values *Val* that includes *PL* and the ranges of all data variables. We are going to speak about *consistent states* only: those are states that

- map program counters to elements of *PL* and
- satisfy mutual exclusion: the sets of resources held by different threads are disjoint.

A thread t *holds* a resource r in a consistent state v if $r \in rsc(v(pc_t))$. A resource r is *busy* in a consistent state v if some thread holds it in v, otherwise r is *available* in v.

Each statement of thread t induces a set of transitions $v \rightarrow_t v'$ where v, v' are consistent states. The transitions induced by the non-`with` statements are straightforward. The statement `with` r `when` ϕ `do` C blocks when r is busy or when ϕ does not hold, otherwise it changes the control flow location to the initial control flow location of C; going out of the critical section is an unconditional change of the control flow location.

Let *init* be the set of initial states corresponding to the initial condition and let the successor map

$$post(S) \;=\; \{v' \mid \exists\, v \in S,\, t \in Tid\colon\, v \rightarrow_t v'\}$$

return the set of all one-step successors of a set of consistent states.

The *syntactic structure* of an RPL program is given by: thread identifiers, control flow locations, values, resources, locals and program counter variables of the threads, proof variables, map *rsc*. A syntactic structure is just a tuple of basic mathematical objects (sets, variables, maps). It also exists independently of an RPL program.

3 Owicki-Gries-Core Proofs

3.1 Lattice of Owicki-Gries-Core Proofs

Now we formalize the core of the Owicki-Gries proof system. We abstract away from the details of logic, handling sets of variable valuations instead of formulas.

Fix an arbitrary syntactic structure.

A *program annotation* (I, M) consists of

- a resource invariant I_r for each resource r, which is a set of valuations of proof variables of r,
- a control flow annotation $M_{p,t}$ for each control flow location p and each thread t, containing valuations of data variables which are local to t or which are proof variables of a resource held at p.

We say that a consistent state v satisfies I_r (resp. $M_{p,t}$), if the projection of v to $ProofVar(r)$ (resp. to local data variables of t and all proof variables of all resources in $rsc(p)$) is in I_r (resp. in $M_{p,t}$).

A program annotation (I, M) *denotes* the set of all consistent states v such that

- for each resource r which is available in v, the state v satisfies I_r, and
- for each thread t, the state v satisfies $M_{v(pc_t),t}$.

We define $\gamma_{\mathrm{OG}}(I, M)$ as the set of all consistent states denoted by (I, M).

Now fix an arbitrary RPL program obeying the given syntactic structure.

An *Owicki-Gries-core proof* of the program is a program annotation that satisfies the following conditions.

- It is *sequentially consistent*, which means the following: consider any transition $v \rightarrow_t v'$ from a control flow location p to a control flow location p' such that v satisfies $M_{p,t}$. There are three cases.
  - The transition does not cross the border of a critical section. Then v' should satisfy $M_{p',t}$.
  - The transition starts a critical section of a resource r. Additionally, assume that v satisfies I_r. Then v' should satisfy $M_{p',t}$.

- The transition finishes a critical section of a resource r. Then v' should satisfy both $M_{p',t}$ and I_r.

– It admits the initial states, i.e., denotes a superset of *init*.

These Hoare-style rules treat critical sections specially. First, on entering a critical section, we additionally assume the resource invariant. Second, on leaving a critical section, we should prove the resource invariant.

Every Owicki-Gries-core proof denotes an inductive invariant. A set of consistent states S is *Owicki-Gries-core provable* if there is an Owicki-Gries-core proof that denotes a subset of S.

Interestingly, the set of program annotations of a fixed program forms a complete lattice with respect to componentwise inclusion order. Restricting this order to the set of Owicki-Gries-core proofs gives a complete lattice of Owicki-Gries-core proofs, even a Moore family. So each program has a unique smallest Owicki-Gries-core proof. This proof denotes the strongest Owicki-Gries-core-provable property. To investigate the power of the method, we will look at the smallest Owicki-Gries-core proofs and strongest Owicki-Gries-core-provable properties.

3.2 Examples

Now we will look at examples of programs with their smallest Owicki-Gries-core proofs. On Readers-Writers, as well as for a whole class of similarly simple programs, all proof methods will have the same precision, on Upper all proof methods will have different precision and on the class SepThreads no precision loss will be observed at all.

We'll use Owicki-style notation. In particular, "`resource` $control(ww, ar, aw)$" means that the resource named *control* contains exactly ww, ar and aw.

Example 1 (Readers-Writers). A number of threads share a file simultaneously: $n > 1$ need reading access and $m > 1$ writing access. Any number of readers may access the file simultaneously, but a writer must have exclusive access. In Fig. 2, all the data variables range over nonnegative integers by default, subtracting a positive value from 0 blocks.

The strongest Owicki-Gries-core-provable property is

$$
\begin{aligned}
\{v \in \mathit{ConsState} \mid\ & v(ww), v(ar) \in \mathbb{N}_0 \ \wedge\ v(aw) \in \{0,1\} \\
& \wedge \forall\, i \in \{1,\dots,n\} : \\
& \qquad (v(pc_{\texttt{reader}_i}) \in \{x\texttt{ready}y, \texttt{read} \mid x \in \{\texttt{start},\texttt{finish}\} \\
& \qquad\ \wedge\ y \in \{\texttt{A},\texttt{B},\texttt{C}\}\}) \\
& \qquad \wedge (v(pc_{\texttt{reader}_i}) = \texttt{startreadB} \Rightarrow v(ww) = 0) \\
& \qquad \wedge (v(pc_{\texttt{reader}_i}) = \texttt{startreadC} \Rightarrow v(ww) = 0 < v(ar)) \\
& \wedge \forall\, j \in \{1,\dots,m\} : \\
& \qquad (v(pc_{\texttt{writer}_j}) \in \{x\texttt{write}y, \texttt{write} \mid x \in \{\texttt{ask},\texttt{start},\texttt{finish}\} \\
& \qquad\ \wedge\ y \in \{\texttt{A},\texttt{B},\texttt{C}\}\}) \\
& \qquad \wedge (v(pc_{\texttt{writer}_j}) = \texttt{askwriteC} \Rightarrow v(ww) > 0) \\
& \qquad \wedge (v(pc_{\texttt{writer}_j}) = \texttt{startwriteB} \Rightarrow v(ar) = v(aw) = 0) \\
& \qquad \wedge (v(pc_{\texttt{writer}_j}) = \texttt{startwriteC} \Rightarrow v(ar) = 0 < v(aw)) \\
& \qquad \wedge (v(pc_{\texttt{writer}_j}) = \texttt{finishwriteC} \Rightarrow v(aw) = 0)\}\,.
\end{aligned}
$$

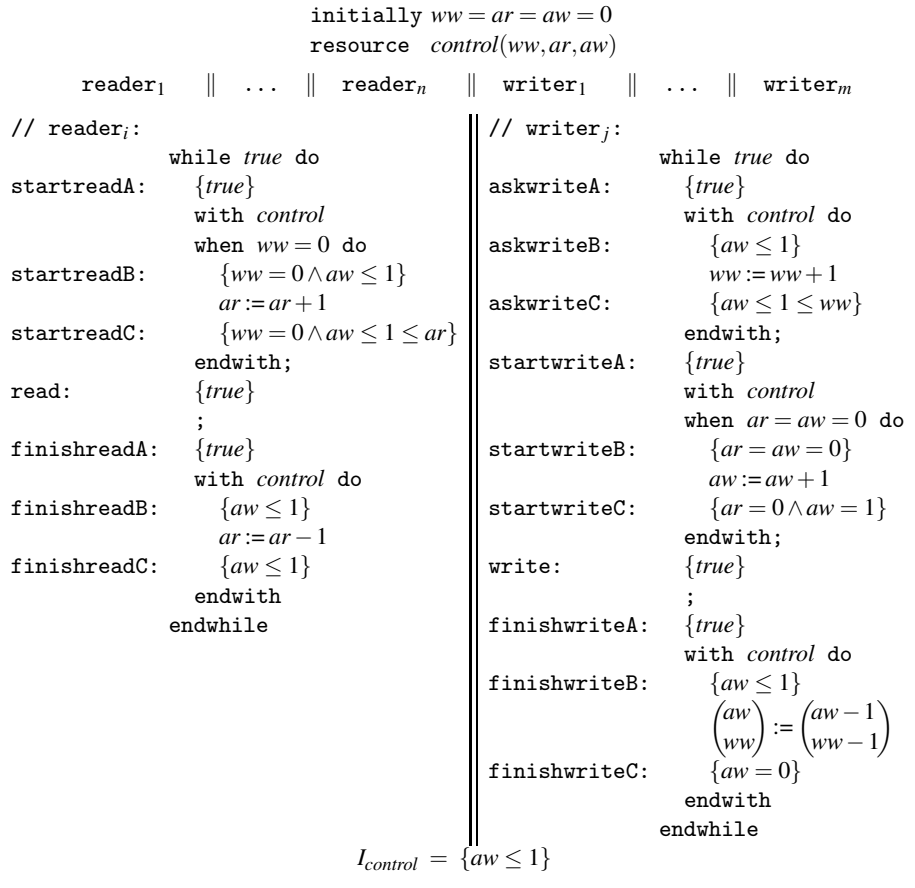


Fig. 2. Program Readers-Writers and its smallest Owicki-Gries-core proof

There are states in this set in which more than one writer is at `write`. Thus no Owicki-Gries-core proof can show that a writer has exclusive access. The smallest Owicki-Gries-core proof can prove a slightly weaker property, namely that the value of *aw*, which tracks the number of writers, never exceeds 1.

Example 2 (progUpper). The program Upper in Fig. 3 will be used for showing differences between the Owicki-Gries-core proofs and the upper bound later. The range of the data variables is $\{0,1\}$. The computation is trivial: no thread can make a step. The majority of the program text serves to create a particular distribution of variables into locals and resources. There are exactly two reachable states, namely the initial ones. The smallest Owicki-Gries-core proof denotes many more states, e.g., $[u \mapsto 0, z \mapsto 1, x \mapsto 0, y \mapsto 1, l \mapsto 0, pc_1 \mapsto \mathtt{A}, pc_2 \mapsto \mathtt{0}]$.

Example 3 (Simple). A program belongs to the class *Simple* if it has at most one resource, and if the resource exists, then it contains no local variables and all its proof variables belong to the resource. Readers-Writers is a family of programs from Simple.

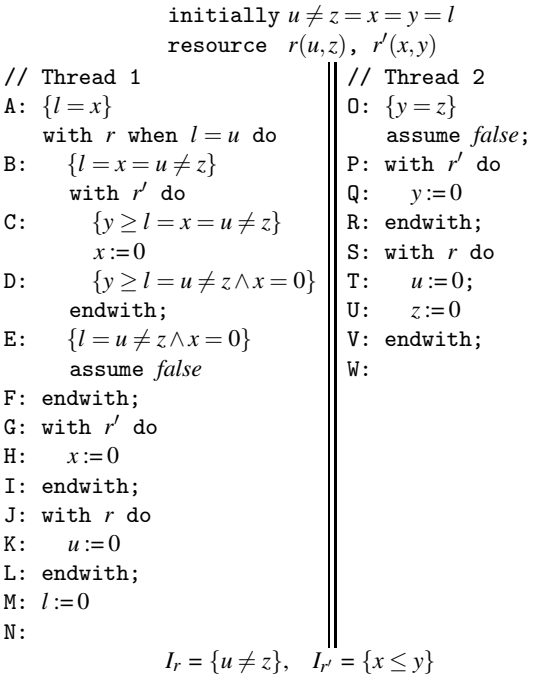


Fig. 3. Program Upper and its smallest Owicki-Gries-core proof. Control flow locations following "assume *false*" are annotated by $\{false\}$ by default.

Example 4 (SepThreads). The class *SepThreads* (Separate Threads) consists of all programs that have no resources and whose initial states are exactly those that satisfy the initial conditions of all the threads (a proper subset of such states is disallowed). For such programs the smallest Owicki-Gries-core proof denotes the set of reachable states.

4 Owicki-Gries Core as Abstract Interpretation

4.1 Owicki-Gries-Core Proofs are the Post-Fixpoints of a Sound Abstract Successor Map

Our first step to try to give a measure of the precision of the Owicki-Gries core will be to cast it in the abstract interpretation framework. An Owicki-Gries-core proof is clearly an abstraction of the set of reachable states of a program. So we will give a characterization of Owicki-Gries core as a post-fixpoint of a sound abstraction of the successor map *post* of an RPL program. The map mimics the application of the Owicki-Gries-core proof conditions.

For a program annotation (I,M), we define the *sound Owicki-Gries-core abstract successor map* $post^{\#}_{\mathrm{OG}}(I,M) = (I',M')$, which applies the sequential consistency once:

- I'_r is the smallest superset of I_r that contains all valuations w of proof variables of r such that there is a transition $v \rightarrow_t v'$ (for some thread t) that exits a critical section

for r, v satisfies $M_{v(pc_t),t}$ and w equals the projection of v' on the proof variables of r;
- $M'_{p',t}$ is the smallest superset of $M_{p',t}$ that contains all valuations w of local data variables of thread t and proof variables of resources held at location p' such that there is a transition $v \rightarrow_t v'$ such that v satisfies $M_{v(pc_t),t}$, v satisfies I_r if the transition starts a critical section for r, $v'(pc_t)=p'$, and w equals the projection of v' to local data variables of t and to the proof variables of resources held at p'.

This operator is, as the name says, sound with respect to the successor map.
Let (I^{init}, M^{init}) be the annotation describing the initial states only:
- I_r^{init} is the set of all valuations of proof variables of r in the initial states;
- $M_{p,t}^{init}$ is the set of valuations of local data variables of thread t in the initial states.

The *Owicki-Gries-core abstract transformer* $F^{\#}_{\mathrm{OG}}(I,M)$ is constructed as the pointwise union of (I^{init}, M^{init}) and $post^{\#}_{\mathrm{OG}}(I,M)$. The set of post-fixpoints of $F^{\#}_{\mathrm{OG}}$ coincides with the set of Owicki-Gries-core proofs; thus the following theorem holds.

Theorem 5 (Equivalence). *The least fixpoint of $F^{\#}_{\mathrm{OG}}$ is the smallest Owicki-Gries-core proof.*

4.2 Characteristic Closures for Owicki-Gries Core?

An elegant way of describing the precision of an approximation of a semantics given in fixpoint form is through the use of *closures*. A closure ρ is a monotone operator that is idempotent ($\rho(\rho(x)) = \rho(x)$) and extensive ($\rho(x) \geq x$). Given a concrete domain $(D, \leq)$ and a monotone function $F\colon D \rightarrow D$, the closure ρ on D defines an approximation of the concrete semantics $lfp(F)$ in the sense that $lfp(F) \leq lfp(\rho \circ F)$. Such a description shows the actual loss of information, as the fixpoints of the closure are exactly the abstract elements that describe the approximation, and applying the closure to one concrete element exactly shows what information is lost for that element.

In our case, we have a concrete domain of sets of consistent states ordered by inclusion, and the semantics is given as the least fixpoint of the successor map *post* over the initial states. Then we exhibited an approximation $post^{\#}_{\mathrm{OG}}$ of that successor map. In order to describe its precision, it would be nice to find a closure ρ such that $post^{\#}_{\mathrm{OG}}$ is exactly the best transformer for $\rho \circ post$. Finding one closure for each program is not difficult (just take the closure with two fixpoints, one being the strongest Owicki-Gries-core-provable property and the other the set of all consistent states), but this would not be very informative. Instead, because the concrete domain is entirely fixed by the syntactic structure of the program, we would like to find a ρ that would be fixed for a given syntactic structure. Alas, the next section shows that it is not possible in general.

5 Absence of Equivalent Characterizations by Closures

The main result of this section is unfortunately negative: assuming we start from a reachable state semantics, there is no way to describe the strongest Owicki-Gries-core proof for a given syntactic structure using closures.

Theorem 6 (No Equivalence). *There is a syntactic structure such that for consistent states defined through the syntactic structure and the concrete domain being the powerset of consistent states, there is no closure ρ on the powerset of consistent states such that for any multithreaded program having the given syntactic structure and for any property S of consistent states we have*

$$S \text{ is Owicki-Gries-core provable} \quad \Leftrightarrow \quad lfp(\lambda x.\, \rho(init \cup post(x))) \subseteq S.$$

One such syntactic structure is given by program Upper from Example 2.

In fact, we can prove an even stronger property, as the proof requires only that ρ is monotone. Even more, the same proof holds even if we restrict the validity of the equivalence to a given transition relation:

Theorem 7. *Under the same syntactic structure as in Thm. 6, there is no family of monotone maps Ξ indexed by transition relations, such that Ξ preserves least fixpoints ($\forall$ transition relations τ_1, τ_2: $lfp(\tau_1) = lfp(\tau_2) \Rightarrow lfp(\Xi_{\tau_1}) = lfp(\Xi_{\tau_2})$), and*

$$S \text{ is Owicki-Gries-core provable} \quad \Leftrightarrow \quad lfp(\Xi_{\lambda x.\, init \cup post(x)}) \subseteq S.$$

Theorem 6 shows that if we start by approximating traces by states and wish to describe the Owicki-Gries-core proof system using closures, we can only hope for bounds framing the proof system. We will provide such bounds in the next two sections. If we are willing to work with closures on sets of traces instead of sets of states, it might be possible to find some equivalence. But such an equivalence cannot be obtained directly by collecting the states of traces obtained from abstract interpretation with a closure. Let $\widetilde{\alpha}$ be the abstraction which associates to a set of traces the set of consistent states appearing in the traces.

Theorem 8. *Under the same syntactic structure as in Thm. 6, there is no monotone operator $\widetilde{\rho}$ on the powerset of traces of consistent states such that for any multithreaded program having the given syntactic structure, set of initial traces $\widetilde{init}$ and the trace extension operator $\widetilde{post}$, and for any property S of consistent states we have*

$$S \text{ is Owicki-Gries-core provable} \quad \Leftrightarrow \quad \widetilde{\alpha}(lfp(\lambda x.\, \widetilde{\rho}(\widetilde{init} \cup \widetilde{post}(x)))) \subseteq S.$$

6 Upper Bound on Precision

Now we will show a closure operator such that the best abstract interpretation of the program with this approximation allows proving a larger set of properties than those provable by the Owicki-Gries core. The approximation will depend only on the syntactic structure, but not on the exact transition relation of a program.

Definition 9 (Owicki-Gries-core annotation closure). *For a given syntactic structure, let $\underline{\rho}(Q)$ be the approximation defined as the set of consistent states v such that:*

- *if a resource r is available in v, then there is some other state in Q*
  - *that coincides with v on the proof variables of r and*
  - *in which r is available;*

– *and for any thread t there is a state in Q that coincides with v on local variables (including the program counter) of t and on the proof variables of the resources held by t in v.*

Then $\underline{\rho}$ is a closure on the powerset of consistent states. We call it the Owicki-Gries-core annotation closure.

The reason why we call this closure an Owicki-Gries-core annotation closure is because it is the closure corresponding to the Galois connection defined by γ_{OG} (which gives the set of consistent states denoted by a program annotation).

Now fix an arbitrary program and let $\underline{\rho}$ be defined by its syntactic structure.

Theorem 10 (Upper bound). *Abstract interpretation with $\underline{\rho}$ is at least as strong as the Owicki-Gries core. Formally:*

$$lfp(\lambda x.\underline{\rho}(init \cup post(x))) \subseteq \textit{the strongest Owicki-Gries-core-provable property}.$$

From the high-level view, the best transformer using this closure is capable of taking into account *global* computation instead of *local* successor computation in the Owicki-Gries core. It is as if before checking sequential consistency we take into account annotations not only of one thread, but of all the threads, gaining precision.

Furthermore, the Owicki-Gries-core annotation closure shows where the information is always lost. For instance, if a syntactic structure says that locals are disjoint among themselves and from all the proof variables of the resources, and if two states outside of critical sections are given, then any combination of the locals and resource variables of those states is in the approximation of those two states.

Example 11 (Readers-Writers). For the program Readers-Writers from Example 1 the least fixpoint of $lfp(\lambda x.\underline{\rho}(init \cup post(x)))$ coincides with the strongest Owicki-Gries-core-provable property. This property is coarser than the set of reachable states. For example, in one execution $\texttt{writer}_1$ can reach `write`, in another execution $\texttt{writer}_2$ can reach `write`, and the initial state satisfies $ww = ar = aw = 0$, so $\underline{\rho}$ produces a combined state where both writers are at `write`, other threads are at their initial locations and all data variables have value 0. Thus, no Owicki-Gries-core proof can restore the dependency between the threads and prove mutual exclusion between the writers.

Example 12 (Upper). For the program Upper from Example 2 abstract interpretation with $\underline{\rho}$ produces the set of reachable states, which is properly included into every Owicki-Gries-core-provable property. The reason for this discrepancy is that locals of one thread and resources held by a different thread overlap. Such an overlap constrains the output of $\underline{\rho}$, but not an Owicki-Gries-core proof. Considering such overlaps actually improves precision!

Example 13 (Simple). For the programs of the class Simple abstract interpretation with $\underline{\rho}$ produces the same result as the smallest Owicki-Gries-core proof. The reason, as we will see, is that abstract interpretation with the lower bound will produce the same result as abstract interpretation with the upper bound.

Example 14 (SepThreads). For the programs of the class SepThreads of Example 4 abstract interpretation with $\underline{\rho}$ produces the same result as the smallest Owicki-Gries-core proof. The reason is that the smallest Owicki-Gries-core proof already denotes the set of reachable states.

7 Lower Bound on Precision

Now we will show a nontrivial Cartesian-like closure such that abstract interpretation with this closure can prove only properties weaker than or equal to the strongest Owicki-Gries-core-provable property. In fact, we will even show two such closures. One closure (namely $\bar{\rho}$) will describe a better lower bound, while the other one (namely ρ_c) is easier to comprehend.

The definitions of both bounds require some preparation.

For a family of sets $\mathscr{X} = \{X_1, \ldots, X_n\}$, let $Part(\mathscr{X})$, called *partition* of $\mathscr{X}$, be the set of all nonempty $Y_1 \cap \ldots \cap Y_n$ where each Y_i is either X_i or its complement $(\bigcup \mathscr{X}) \setminus X_i$ $(1 \leq i \leq n)$. Elements of a partition are called *blocks*.

Let us fix an arbitrary syntactic structure. Our lower bounds will depend on the syntactic structure but be the same for all the programs that obey this syntactic structure. Let *RL* be the family of sets containing all resources r and all sets of locals $Local_t$ for all threads t as elements. Let $\widetilde{RL}$ be the partition of *RL*.

For example, for the syntactic structure of Upper the set *RL* has exactly four elements: the locals of the first thread $\{l, x, pc_1\}$, the locals of the second thread $\{y, z, pc_2\}$, the resource $r = \{u, z\}$, and the resource $r' = \{x, y\}$. The corresponding partition has exactly six blocks: $\{u\}$, $\{x\}$, $\{y\}$, $\{z\}$, $\{l, pc_1\}$, $\{pc_2\}$.

7.1 Simple Cartesian Closure

The simpler approximation is defined as follows.

Definition 15 (Cartesian closure). *Given a set of consistent states Q, the* Cartesian *approximation ρ_c returns all the consistent states from the product of projections of Q onto the blocks in $\widetilde{RL}$.*
In other words, $\rho_c(Q)$ contains exactly those consistent states v such that for each block there is a state $\tilde{v}$ in Q that agrees with v on the variables of the block.

As the name says, ρ_c is a closure.

This approximation breaks all the dependencies between the blocks and retains all dependencies inside a block.

For example, for the syntactic structure of the program Upper the variables l and pc_1 belong to the same block. Thus, if in a set of states every state at some fixed control flow location satisfies $l = 0$, each state from the Cartesian approximation of this set will also satisfy $l = 0$ for that control flow location.

Abstract interpretation with ρ_c generates a property which is always weaker than or equal to the strongest Owicki-Gries-core-provable property.

7.2 More Precise Lower Bound Closure

The following definition strengthens ρ_c in two ways. Firstly, we impose restrictions on $\tilde{v}$ from the definition of ρ_c. Such restrictions will depend on the block. Secondly, we look at the prophecy variables: those are variables which are never written and which don't belong to a resource. Prophecy variables form a separate block; now we restore all the dependencies between this block and those locals of any thread that are not resource variables.

Definition 16 (Lower Bound closure). *Given a set of consistent states Q, its approximation $\bar{\rho}(Q)$ contains exactly those consistent states v such that both of the following conditions are satisfied.*

- *For every block in $\widetilde{RL}$ that is contained in a resource,*
  - *if the resource is available in v, then there is some $\tilde{v} \in Q$*
    - *in which the resource is also available*
    - *and which coincides with v on the block;*
  - *if some thread holds the resource in v, then there is some $\tilde{v} \in Q$ which coincides with v on all the variables*
    - *that are local to this thread but do not belong to any resource, or*
    - *that belong to the block.*
- *For every thread there is a state in Q that coincides with v on each variable*
  - *that is a local variable of the thread but*
  - *does not belong to any resource.*

As the name says, $\bar{\rho}$ is a closure. It is at least as precise as ρ_c. Both closures do not depend on the exact transition relation of a program. The closure $\bar{\rho}$ induces the tightest lower bound we could prove. It shows the dependencies that are always retained, creating a basis for the construction of future refinement algorithms (possibly following [11]).

Now fix an arbitrary RPL program that has the assumed syntactic structure.

The proof of the lower bound relies on several claims about the strongest property provable by abstract interpretation with $\bar{\rho}$. The following claim is the most important one.

Lemma 17. *Let Q be the strongest property provable by abstract interpretation with $\bar{\rho}$. Consider a transition of a thread t from a consistent state v to a consistent state v', let the transition start a critical section. Let $\tilde{v} \in Q$ such that v agrees with $\tilde{v}$ on the locals of t and on the variables of the resources held by t before the transition. Let $\hat{v} \in Q$ such that v agrees with $\hat{v}$ on the variables of the resource being acquired. Then there is a state in Q that agrees with v' on the locals of thread t and on the variables of the resources held by the thread after the transition.*

The lower bound theorem follows from the lemma.

Theorem 18 (Lower bound). *The core of Owicki-Gries can prove at least as many properties as abstract interpretation with $\bar{\rho}$ or ρ_c. Formally:*
the strongest Owicki-Gries-core–provable property $\subseteq$ $lfp(\lambda x.\, \bar{\rho}(init \cup post(x)))$
$\subseteq$ $lfp(\lambda x.\, \rho_c(init \cup post(x)))$.

Example 19 (Readers-Writers). For the program Readers-Writers from Example 1 abstract interpretation with ρ_c produces the set of all consistent states where readers are at `...read...`, writers are at `...write...`, and *ww*, *ar*, *aw* are nonnegative. Owicki-Gries-core and abstract interpretation with $\bar{\rho}$ can prove stronger properties, e.g., that at location `askwriteC` the value of *ww* is positive. Intuitively, when a resource is busy, Cartesian abstraction always breaks the dependency between the resource variables and the control flow, while the Owicki-Gries core sometimes retains the dependency.

Example 20 (Upper). For the program Upper from Example 2 abstract interpretation using either ρ_c or $\bar{\rho}$ produces the same result: the set of all states such that the first thread is at any location between `A` and `E`, the second thread is at `O` and all data variables take arbitrary values from $\{0,1\}$. The Owicki-Gries core can prove stronger properties; for instance, it can show that at location `E` the value of x is zero. Intuitively, the dependency between resource variables and control flow is always broken in Cartesian abstraction but is sometimes retained in the Owicki-Gries core.

Example 21 (Simple). For the programs from the class Simple from Example 3 the approximations $\bar{\rho}$ and $\underline{\rho}$ are so close to each other that abstract interpretations with both produce the same property. Since they define the lower and upper bounds on the precision of the Owicki-Gries core, the Owicki-Gries core can prove exactly the same properties as those provable by abstract interpretation with any of the two approximations. If a program from Simple does not have the empty resource, then $\bar{\rho}$ and $\underline{\rho}$ coincide exactly, approximating a set of states Q by the set of all consistent states v such that both of the following conditions hold.

- If there is a resource and it is available in v, there is some state in Q that coincides with v on the resource and in which the resource is available.
- For each thread there is a state in Q which coincides with v on the variables of the thread and, if the resource is present and is held, on the variables of the resource.

Example 22 (SepThreads). For the programs from the class SepThreads from Example 4 the approximations $\bar{\rho}$ and $\underline{\rho}$ coincide. Since they define the lower and upper bounds on the precision of the Owicki-Gries core, the Owicki-Gries core can prove exactly the same properties as those provable by abstract interpretation with any of the two approximations. Due to the absence of any thread interactions and independence of initial states of the threads, the mentioned methods can prove the strongest inductive property, namely the set of states reachable from the initial ones. Informally spoken, all dependencies between the locals of each thread are retained.

8 Related Work

Historically, conditional critical regions were introduced in [8]. The thesis [15] of Owicki and her paper with Gries [16] describe the original proof method for RPL. Modular reasoning about RPL has not been characterized in terms of abstract interpretation via closures [5] so far.

For general multithreaded programs (i.e. without separation of data into resources), Owicki-Gries-style reasoning without auxiliary variables is equivalent to multithreaded Cartesian abstraction. The result was first mentioned without proof in [6], and the proof appears in [11].

Clarke [3] has considered a subset of integer RPL programs with only one resource, where, roughly, only additions of constants inside the critical sections are allowed, and the property to be checked is either mutual exclusion of PV-semaphores or deadlock freedom. With a predefined choice of integer auxiliary variables, the least fixpoint of a particular functional is a resource invariant that precisely tells whether the property holds or not. Two overapproximations are given: a fixed-formula resource invariant

and an invariant computed by a polyhedral analysis with widening. Our work, on the contrary, does not impose any restrictions on the program form. Our results hold for even more programs than the RPL ones, e.g., where the critical sections are not well-nested.

The work of Owicki on RPL is the basis of a variety of modular programming languages equipped with proof methods of different degrees of completeness and automation.

Concurrent separation logic (CSL) [14] equips RPL with separation logic as a formula language. CSL is also incomplete without the rule of auxiliary variables, so the question of precision arises. Removing secondly important features of CSL for the sake of clarity (as in [2]) and considering variables in the heap makes our lower bound also apply to such CSL versions.

Chalice [10] is a language for verification of object-oriented concurrent programs with heap, equipped with an RPL-like proof system. Due to the powerful permission system, the proof system is in general stronger than that of Owicki, so our lower bound on precision carries over to Chalice for programs that can be directly represented both in RPL and in Chalice.

VCC [13] is a verifier for multithreaded C. When accessing structures in a lock-based manner, VCC requires the user to provide invariants of C `structs`. On obtaining ownership of a `structure`, the resource invariant is assumed; on relinquishing ownership, the resource invariant has to be reestablished. Ghost contracts of lock-manipulating functions control the ownership transfer. Our lower bound applies to VCC as well.

9 Discussion

9.1 Challenges

Discussing the precision loss reveals several open problems.

Example 20 shows a gap between the accuracy of the Owicki-Gries core and the lower bound. Can the lower bound closure be strengthened?

The main inequivalence result assumes that the concrete domain is the powerset of consistent states. For the powerset of traces, we only know inequivalence for a subclass of abstract interpretations. Is there an exact characterization of the Owicki-Gries core by abstract interpretation on the powerset of traces?

We have shown what variable dependencies does the Owicki-Gries core break. Can these dependencies be restored on demand? Is there an automatic counterexample-guided abstraction refinement of the Owicki-Gries core, perhaps based on auxiliary variables [4], unions of Cartesian products [11], or abstract threads [9]?

Can one characterize the precision loss of the Owicki-Gries core by completeness notions of [7]?

Can one formalize CSL in abstract interpretation in a way that would reveal the involved approximation?

9.2 Conclusion

We have examined a modular method (Owicki-Gries core) for proving safety properties of a widely-used class of multithreaded programs.

The considered class contains structured programs in which shared data are partitioned into resources and are accessed only in critical sections that ensure mutually exclusive access to resources. The method provides a clean basis for other more sophisticated proof methods like Concurrent Separation Logic, Chalice, or VCC. The Owicki-Gries core is polynomial in the number of threads, but without manually adding auxiliary variables it cannot prove many properties of concurrent programs.

The Owicki-Gries core is, intuitively, expected to succeed for properties whose dependence on thread coupling is low, and is expected to fail if complicated thread interactions have to be analyzed. We have made this notion precise, providing a characterization of the set of Owicki-Gries-core-provable properties. We have demonstrated an abstract transformer corresponding to the Owicki-Gries core: the least fixpoint of the abstract transformer denotes exactly the strongest Owicki-Gries-core-provable property. To quantify the loss of precision inherent to modularity, we have provided a superset and a subset of Owicki-Gries-core-provable properties, described by abstract interpretations with closure operators that depend on the syntactic structure of the program only. These bounds coincide for a class of simple programs. We have also shown a principal inability to provide an exact characterization of the set of properties in terms of closures that depend only on the syntactic structure.

Acknowledgements. We thank Byron Cook and Microsoft Research Cambridge for support in the initial stages of the work. The work was also partially supported by the Verisoft project of the German science foundation DFG. We thank Josh Berdine and Viktor Vafeiadis for helpful comments and discussions. We are grateful to Springer for the assistance during typesetting.

References

1. Barthe, G., Hermenegildo, M. (eds.): VMCAI 2010. LNCS, vol. 5944. Springer, Heidelberg (2010)
2. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS, pp. 366–378. IEEE Computer Society (2007)
3. Clarke, E.M.: Synthesis of resource invariants for concurrent programs. ACM Trans. Program. Lang. Syst. 2(3), 338–358 (1980)
4. 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)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
6. Cousot, R.: Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles. PhD thesis, Institut national polytechnique de Lorraine (1985)
7. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. Journal of the ACM 47(2), 361–416 (2000)
8. Hoare, C.A.R.: Towards a theory of parallel programming. In: Hoare, C.A.R., Perrott, R.H. (eds.) Operating System Techniques, pp. 61–71. Academic Press (1972)
9. Lahiri, S.K., Malkis, A., Qadeer, S.: Abstract threads. In: Barthe and Hermenegildo [1], pp. 231–246
10. Leino, K.R.M.: Verifying concurrent programs with Chalice. In: Barthe and Hermenegildo [1], p. 2

11. Malkis, A.: Cartesian Abstraction and Verification of Multithreaded Programs. PhD thesis, Albert-Ludwigs-Universität Freiburg (February 2010)
12. Malkis, A., Mauborgne, L.: On the strength of Owicki-Gries for resources. Technical report, IMDEA Software Institute (2011), http://software.imdea.org/~alexmalkis/onTheStrengthOfOwickiGriesForResources_techrep.ps
13. Moskal, M., Schulte, W., Cohen, E., Hillebrand, M.A., Tobies, S.: Verifying C programs: A VCC tutorial, MSR Redmond, EMIC Aachen (2011)
14. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
15. Owicki, S.S.: Axiomatic Proof Techniques For Parallel Programs. PhD thesis, Cornell University, Department of Computer Science, TR 75-251 (July 1975)
16. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM 19(5), 279–285 (1976)

Solving Recursion-Free Horn Clauses over LI+UIF

Ashutosh Gupta[1,2], Corneliu Popeea[2], and Andrey Rybalchenko[2]

[1] IST Austria
[2] Technische Universität München

Abstract. Verification of programs with procedures, multi-threaded programs, and higher-order functional programs can be effectively automated using abstraction and refinement schemes that rely on spurious counterexamples for abstraction discovery. The analysis of counterexamples can be automated by a series of interpolation queries, or, alternatively, as a constraint solving query expressed by a set of recursion free Horn clauses. (A set of interpolation queries can be formulated as a single constraint over Horn clauses with linear dependency structure between the unknown relations.) In this paper we present an algorithm for solving recursion free Horn clauses over a combined theory of linear real/rational arithmetic and uninterpreted functions. Our algorithm performs resolution to deal with the clausal structure and relies on partial solutions to deal with (non-local) instances of functionality axioms.

1 Introduction

Constraint solving is a vehicle of software verification that provides symbolic reasoning techniques for dealing with assertions describing program behaviors. In particular, abstraction and refinement techniques greatly benefit from applying constraint solving, where interpolation techniques [1, 2, 3, 5, 6, 11, 12, 13, 16, 17, 18, 19] play a prominent role today. Roughly, interpolation computes an assertion that separates two mutually unsatisfiable assertions and only refers to their shared symbols.

Certain abstraction refinement tasks cannot be *directly* expressed as an interpolation question. For example, abstraction refinement for imperative programs with procedures [12] and for higher order functional programs [14, 20], require additional pre-processing that splits discovered spurious counterexamples in multiple ways and applies interpolation on each splitting. Alternatively, as exemplified by an abstraction refinement procedure for multi-threaded programs [9], this preprocessing and series of interpolation computations can be expressed using a single constraint that consists of a finite set of recursion-free Horn clauses interpreted over the logical theory that is used to describe program behaviors.

In this paper, we present an algorithm for solving Horn clauses over a combination of linear rational/real arithmetic, uninterpreted functions and queries.

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 188–203, 2011.

Our algorithm opens new possibilities for the development of abstraction refinement schemes by providing the verification method designer an expressive, declarative way to specify what the refinement procedure needs to compute using Horn clauses. Several existing abstraction refinement schemes can directly benefit from our algorithm, e.g., for programs with procedures [11, 12], for multi-threaded programs [9], and for higher-order functional programs [14, 20, 21].

Related Work. In [9] we presented an algorithm that deals with recursion-free Horn clauses over linear real/rational arithmetic. Here, we present an extension with uninterpreted functions.

Technically, our treatment of uninterpreted functions can be seen as a generalization of partial interpolants [17] to partial solutions for recursion-free Horn clauses, i.e., clauses that do not have cyclic dependencies between the occurring queries. Our algorithm follows a general scheme of combining interpolation procedures for different theories [8, 22].

The following example illustrates the relation to interpolation. First, we consider an interpolation question for a pair of mutually unsatisfiable assertions $a(x, y)$ and $b(y, z)$ in a logical theory. An interpolant is an assertion $I(y)$ such that $I(y)$ is a logical consequence of $a(x, y)$, $I(y)$ and $b(y, z)$ are mutually unsatisfiable, and $I(y)$ only contains non-theory constants that are shared by $a(x, y)$ and $b(y, z)$, which is y in our example. Now, we present our re-formulation of interpolation as a constraint solving question for constraints given by recursion-free Horn clauses. We introduce a relation $Q_I(y)$ that represents an interpolant that we want to compute. We represent the interpolation conditions by the following two Horn clauses: $a(x, y) \rightarrow Q_I(y)$ and $Q_I(y) \wedge b(y, z) \rightarrow \mathit{false}$. Any interpretation of $Q_I(y)$ that only refers to y (and theory constants) is an interpolant for $a(x, y)$ and $b(y, z)$. In Section 6, we discuss the relation of this paper with [17] in more detail.

Horn clauses with more that two unknown queries in the body do not directly correspond to interpolation problems. For example, we consider two relations $Q_I(y)$ and $Q_J(y)$ that represent assertions we want to compute together with the Horn clauses $a(x, y) \rightarrow Q_I(y)$, $b(y, z) \rightarrow Q_J(y)$, and $Q_I(y) \wedge Q_J(y) \rightarrow \mathit{false}$. Solving these clauses using interpolation requires two invocations of an interpolation procedure (i.e., interpolation between $a(x, y)$ and $b(y, z)$ determines $Q_I(y)$ interpolation between $b(y, z)$ and $Q_I(y)$ determines $Q_J(y)$) that we would like to avoid for efficiency considerations. Furthermore, by computing $Q_I(y)$ and $Q_J(y)$ one after the other it is not evident how to compute solutions satisfying certain preference conditions, e.g., where all constrants are within predefined bounds (such conditions are useful for abstraction refinement, as shown by the FOCI procedure [15]).

Organization. Section 2 illustrates our algorithm. Section 3 provides formal definitions. We present the solving algorithm in Section 4 and discuss its correctness and complexity in Section 5. Section 6 concludes and clarifies the connection of our algorithm to interpolation procedures.

We also provide an extended version of this paper [10]. In this extended version, Appendix A illustrates how Horn clauses can be used for abstraction

refinement of procedural and multi-threaded programs, Appendix B contains a complete example execution of our algorithm, and Appendix C presents proofs of the key theorems.

2 Illustration

In this section, we shall illustrate our proposed algorithm by solving an example set of Horn clauses. The example set of Horn clauses $\mathcal{HC}$ is presented in Figure 1(a) and consists of three clauses. Our algorithm is looking for solutions to the two symbols $S(\mathtt{t},\mathtt{u},\mathtt{v})$ and $E(\mathtt{t},\mathtt{u})$ that we name *queries*. To obtain solutions over the domain of linear arithmetic and uninterpreted functions, our algorithm proceeds following three steps.

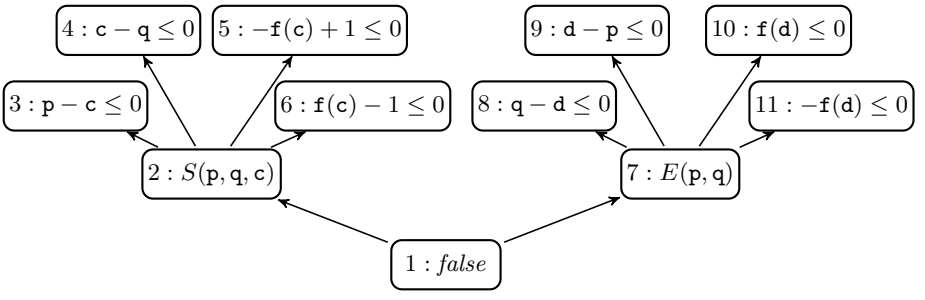
Resolution Tree. Our solving algorithm starts by constructing from $\mathcal{HC}$ a resolution tree R shown in Figure 1(b). We label nodes of R with indices for easy reference. From the first clause, the algorithm constructs the subtree rooted at label 2. In this subtree, we have edges between the node corresponding to the head of the clause (labeled 2) and the nodes corresponding to the body of the clause (labeled 3–6). A second subtree rooted at the node labelled 7 is constructed from the second clause. With the appearance of the queries $S(\mathtt{t},\mathtt{u},\mathtt{v})$ and $E(\mathtt{t},\mathtt{u})$ in the body of the third clause from $\mathcal{HC}$, the two previously constructed subtrees are extended in a tree with the root labeled corresponding to the clause head, $(1 : \mathit{false})$. The extension of the subtrees leads to the variables occurring in these subtrees to be renamed to a common set of variables $\mathtt{p},\mathtt{q},\mathtt{c}$. Note that, the set of clauses $\mathcal{HC}$ is satisfiable, and, consequently, the conjunction of the predicates from the leaves of the resolution tree is unsatisfiable.

Proof Tree. Next, our algorithm constructs a proof tree that proves unsatisfiability of the constraints from the leaves of the resolution tree. For the resolution tree from Figure 1(b), our algorithm computes the proof tree P shown in Figure 1(c). A linear combination rule is applied to derive the constraint $(\mathtt{c}-\mathtt{d}\leq 0)$ from the premises $(\mathtt{c}-\mathtt{q}\leq 0)$ and $(\mathtt{q}-\mathtt{d}\leq 0)$. The linear combination rule is also used to derive $(\mathtt{d}-\mathtt{c}\leq 0)$ from the premises $(\mathtt{p}-\mathtt{c}\leq 0)$ and $(\mathtt{d}-\mathtt{p}\leq 0)$. A congruence rule is used to relate function symbols applied to equivalent arguments. This rule derives $(\mathtt{f}(\mathtt{c})-\mathtt{f}(\mathtt{d})\leq 0)$ from the premises $(\mathtt{c}-\mathtt{d}\leq 0)$ and $(\mathtt{d}-\mathtt{c}\leq 0)$. Lastly, $(1\leq 0)$ is derived by applying the linear combination rule on three premises, $(\mathtt{f}(\mathtt{d})\leq 0)$, $(\mathtt{f}(\mathtt{c})-\mathtt{f}(\mathtt{d})\leq 0)$, and $(-\mathtt{f}(\mathtt{c})+1\leq 0)$.

Partial and Final Solutions. The proof tree P explicates the inference rules and the order in which to apply them to derive the false constraint $(1\leq 0)$. The main idea behind our solving algorithm is to apply corresponding inference rules in the same order to derive a solution for the Horn clauses. We obtain an annotated proof tree (see Figure 1(d)) where for each of the premises used in P , our algorithm creates one tree with the same number of nodes as R .

$$\mathcal{HC} = \{\ \forall \mathtt{p},\mathtt{q},\mathtt{c} : \mathtt{p} \le \mathtt{c} \wedge \mathtt{c} \le \mathtt{q} \wedge -\mathtt{f}(\mathtt{c}) + 1 \le 0 \wedge \mathtt{f}(\mathtt{c}) - 1 \le 0 \rightarrow S(\mathtt{p},\mathtt{q},\mathtt{c}),$$
$$\forall \mathtt{r},\mathtt{s},\mathtt{d} : \mathtt{s} \le \mathtt{d} \wedge \mathtt{d} \le \mathtt{r} \wedge \mathtt{f}(\mathtt{d}) \le 0 \wedge -\mathtt{f}(\mathtt{d}) \le 0 \rightarrow E(\mathtt{r},\mathtt{s}),$$
$$\forall \mathtt{t},\mathtt{u},\mathtt{v} : S(\mathtt{t},\mathtt{u},\mathtt{v}) \wedge E(\mathtt{t},\mathtt{u}) \rightarrow \mathit{false}\ \}$$

(a)



(b)

$$\dfrac{\dfrac{}{\mathtt{f}(\mathtt{d}) \le 0} \quad \dfrac{\dfrac{\mathtt{c}-\mathtt{q} \le 0 \quad \mathtt{q}-\mathtt{d} \le 0}{\mathtt{c}-\mathtt{d} \le 0} \quad \dfrac{\mathtt{p}-\mathtt{c} \le 0 \quad \mathtt{d}-\mathtt{p} \le 0}{\mathtt{d}-\mathtt{c} \le 0}}{\mathtt{f}(\mathtt{c}) - \mathtt{f}(\mathtt{d}) \le 0} \quad \dfrac{}{-\mathtt{f}(\mathtt{c}) + 1 \le 0}}{1 \le 0}$$

(c)

$$\dfrac{\dfrac{}{\mathtt{f}(\mathtt{d}) \le 0[\ldots]} \quad \dfrac{\dfrac{\mathtt{c}-\mathtt{q} \le 0[\Pi_1] \quad \mathtt{q}-\mathtt{d} \le 0[\Pi_2]}{\mathtt{c}-\mathtt{d} \le 0[\Pi_3]} \quad \cdots}{\mathtt{f}(\mathtt{c}) - \mathtt{f}(\mathtt{d}) \le 0[\ldots]} \quad \dfrac{}{-\mathtt{f}(\mathtt{c}) + 1 \le 0[\ldots]}}{1 \le 0[\Pi]}$$

(d)

Fig. 1. (a) A set of Horn clauses $\mathcal{HC}$. (b) Corresponding resolution tree R . (c) Proof of unsatisfiability P for the constraints from the leaves of the resolution tree. For abbreviation, we did not mark nodes of subtree of $\mathtt{f}(\mathtt{c}) - \mathtt{f}(\mathtt{d}) \le 0$ with the applied proof rules. (d) A part of the annotated proof tree. The partial solutions Π_1, Π_2, Π_3, and Π are presented in Figure 2.

We call these trees, which are annotated with formulas that will be explained next, partial-solution trees.

The tree Π_1 corresponds to the premise $(\mathtt{c} - \mathtt{q} \le 0)$, Π_2 corresponds to the premise $(\mathtt{q} - \mathtt{d} \le 0)$ and both trees are shown in Figure 2. Two or more premises are used to derive a new fact in the proof tree and, likewise, two or more corresponding partial-solution trees are used to derive a new tree using a

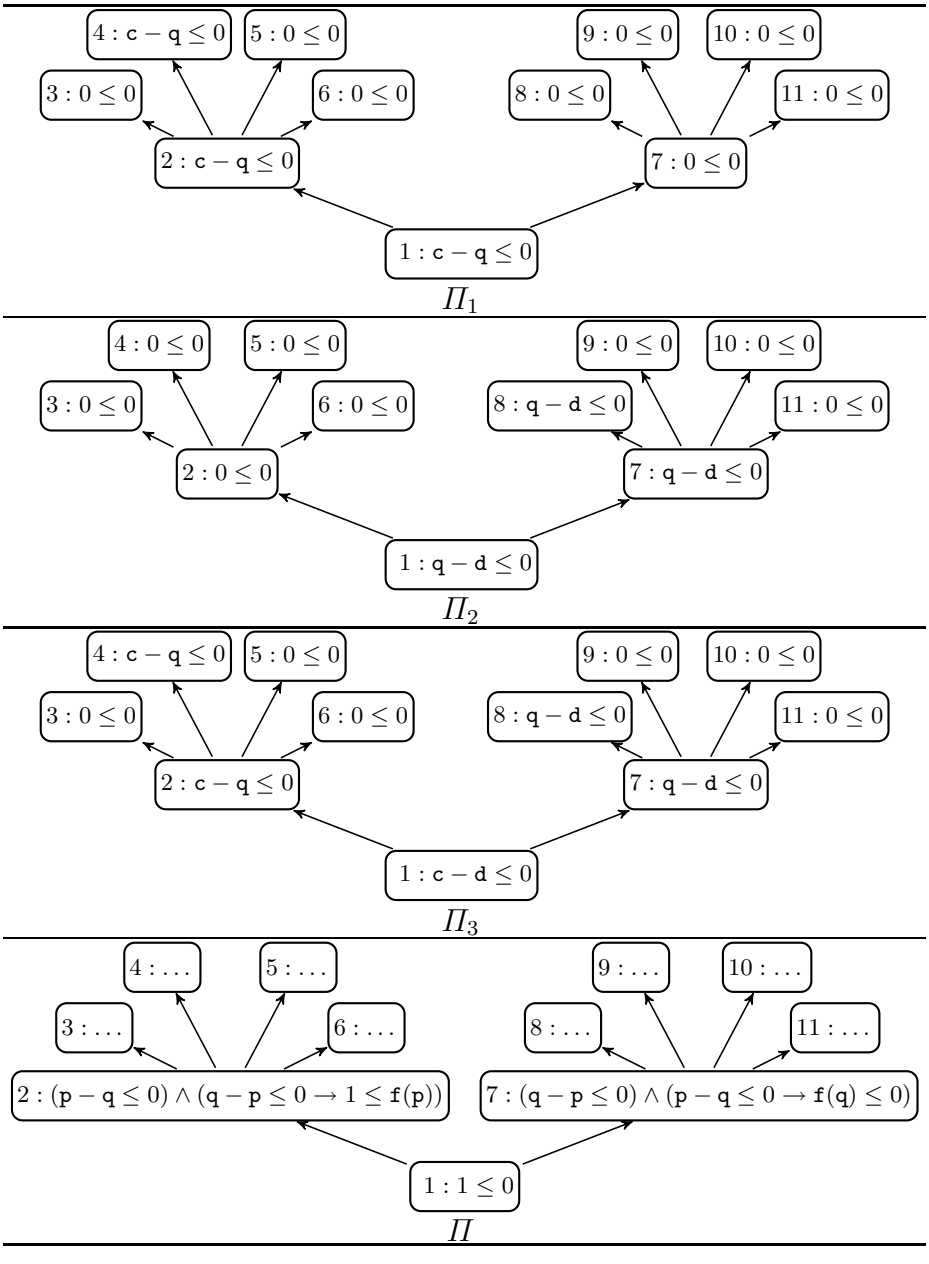


Fig. 2. Four partial-solution trees Π_1 , Π_2 , Π_3 , and Π . Π_1 and Π_2 are derived from the nodes $(\mathtt{c} - \mathtt{q} \leq 0)$ and $(\mathtt{q} - \mathtt{d} \leq 0)$ from the proof tree P from Figure 1(d). Π_3 is obtained by applying a combination rule to Π_1 and Π_2. Π annotates the false constraint $(1 \leq 0)$ from P and the final solution of $\mathcal{HC}$ can be derived from Π . In particular, the nodes labeled "2" and "7" contain the solutions for $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$ and $E(\mathtt{p}, \mathtt{q})$, respectively.

specific inference rule. The two trees Π_1 and Π_2 shown in the top part of Figure 2 are combined using a rule corresponding to the arithmetic combination rule. The rule takes a pair of corresponding nodes, one from Π_1 and one from Π_2 , and computes a node in the resulting tree Π_3 . For the node labeled ($2 : \mathtt{c} - \mathtt{q} \leq 0$) from Π_1 and the node labeled ($2 : 0 \leq 0$) from Π_2, the algorithm adds the two constraints and creates a node labeled ($2 : \mathtt{c} - \mathtt{q} \leq 0$) in Π_3 . Similarly, the nodes labeled ($1 : \mathtt{c} - \mathtt{q} \leq 0$) and ($1 : \mathtt{q} - \mathtt{d} \leq 0$) are used to obtain a node labeled ($1 : \mathtt{c} - \mathtt{d} \leq 0$) in Π_3 .

Following the derivation of the proof tree P, inference rules are used to combine partial-solution trees until a final-solution tree corresponding to the rule applied at the bottom of the proof tree. The final-solution tree is Π and is shown in Figure 2. The node labeled "2" contains the solution for $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$ and it can be simplified to $S(\mathtt{p}, \mathtt{q}, \mathtt{c}) = (\mathtt{p} < \mathtt{q} \vee \mathtt{p} \leq \mathtt{q} \wedge \mathtt{f}(\mathtt{p}) \geq 1)$. The solution from the node labeled "7" can be simplified to $E(\mathtt{p}, \mathtt{q}) = (\mathtt{p} > \mathtt{q} \vee \mathtt{p} \geq \mathtt{q} \wedge \mathtt{f}(\mathtt{p}) \leq 0)$. The solutions obtained for $S(\mathtt{p}, \mathtt{q}, \mathtt{c})$ and $E(\mathtt{p}, \mathtt{q})$ indeed satisfy the set of Horn clauses $\mathcal{HC}$ from Figure 1(a).

3 Recursion-Free Horn Clauses

This section presents auxiliary definitions together with the syntax and semantics of recursion-free Horn clauses over linear arithmetic, uninterpreted functions, and queries.

Syntax. We assume countable sets of *variables* $\mathcal{V}$, with $v \in \mathcal{V}$, *function symbols* $\mathcal{F}$, with $f \in \mathcal{F}$, and *predicate symbols* $\mathcal{P}$, with $p \in \mathcal{P}$. Let the arity of function and predicate symbols be encoded in their names. In addition, we assume a set of *number symbols* $\mathcal{N}$, with $\{0, n\} \subseteq \mathcal{N}$, and an inequality symbol $\leq$. Then, we define:

terms	$\ni t ::= n \mid nv \mid t + t \mid f(t, \ldots, t)$	bodies	$\ni b ::= a \mid q \mid b \wedge b$
atoms	$\ni a ::= t \leq 0$	heads	$\ni h ::= a \mid q \mid \mathit{false}$
queries	$\ni q ::= p(v, \ldots, v)$	Horn clauses	$\ni s ::= b \rightarrow h$

Without loss of generality, as justified later, we assume that all variables that occur in a query are distinct.

A set of Horn clauses defines a binary *dependency* relation on predicate symbols. A predicate symbol $p \in \mathcal{P}$ *depends* on a predicate symbol $p_i \in \mathcal{P}$ if there is a Horn clause $\cdots \wedge p_i(\ldots) \wedge \cdots \rightarrow p(\ldots)$, i.e., when p appears in the head of a clause that contains p_i in its body. A set of Horn clauses is *recursion-free* if the corresponding dependency relation does not contain any cycles. A set of Horn clauses is *tree-like* if the corresponding dependency relation defines a tree-like graph, i.e., when 1) each predicate symbol appears at most once in the set of bodies and at most once in the set of heads of the given clauses, 2) there is no clause with an atom in its head, 3) there is one clause whose head is *false*. For example, the set of clauses $\{p(v_1) \wedge p(v_2) \rightarrow q(v_1, v_2), q(v_3, v_4) \rightarrow \mathit{false}\}$ is not

tree-like since the predicate symbol p appears more than once in the body of the first clause.

For the rest of the presentation, we consider a finite set of Horn clauses $\mathcal{HC}$ that satisfies the following conditions. First, we assume that each variable occurs in at most one clause and that all variables occurring in a query are distinct. These assumptions simplify our presentation and can be established by an appropriate variable renaming and additional (in)equality constraints. Furthermore, we assume that $\mathcal{HC}$ is recursion-free and tree-like. The recursion-free assumption is critical for ensuring termination of the solving algorithm presented in this paper. The tree-like assumption simplifies our presentation without imposing any restrictions on the algorithm's applicability. Any finite set of recursion-free clauses can be transformed into the tree-like form. The solution for the computed tree-like form can be translated into the solution for the original set of clauses.

Finally, we define *constraints* together with a *conjunctive constraint* fragment below.

$$\text{constraints} \ni c ::= a \mid \neg c \mid c \wedge c \mid c \vee c \quad \text{conjunctive constraints} \ni \hat{c} ::= a \mid \hat{c} \wedge \hat{c}$$

Auxiliary Definitions. We assume the following standard functions. For dealing with trees, let $nodes(T)$ be the nodes of a tree T, $root(T)$ be the root node of T, $leaves(T)$ be the leaves of T, and $subtree(o, T)$ be the subtree of T rooted in its node o. Furthermore, let $subterms(C)$ be the subterms occurring in a constraint C and $atoms(C)$ be the atoms occurring in C. Let $sym(t)$ be the variables and uninterpreted function symbols occurring in a term t.

Let $match(p(v_1, \ldots, v_n), p'(v'_1, \ldots, v'_m))$ return a substitution $\{v_1 \mapsto v'_1, \ldots, v_n \mapsto v'_m\}$ if $p = p'$ (and hence $n = m$). Thus, if a substitution σ is the result of $match(p'(v_1, \ldots, v_n), p'(v'_1, \ldots, v'_m))$ then $p'(v_1, \ldots, v_n)\sigma = p'(v'_1, \ldots, v'_m)$, i.e., by applying the substitution we equate the queries. For example, $match(p_1(v_1), q(v_2, v_3))$ is not defined, and $match(q(v_1, v_2), q(v_3, v_4)) = \{v_1 \mapsto v_3, v_2 \mapsto v_4\}$. We assume a canonical extension of the unifier application to constraints and their combination into sequences and sets.

Given two substitutions $\sigma_1 = \{v_1 \mapsto v'_1, \ldots, v_n \mapsto v'_n\}$ and $\sigma_2 = \{w_1 \mapsto w'_1, \ldots, w_m \mapsto w'_m\}$ over disjoint domains, i.e., $\{v_1, \ldots, v_n\} \cap \{w_1, \ldots, w_m\} = \emptyset$, we define a *combined* substitution $\sigma_1 + \sigma_2 = \{v_1 \mapsto v'_1, \ldots, v_n \mapsto v'_n, w_1 \mapsto w'_1, \ldots, w_m \mapsto w'_m\}$.

Semantics. Let $\models$ be the (logical) satisfaction relation for our constraints in the combined theory of linear real/rational arithmetic and uninterpreted functions. We write $\models c$ when c is a valid constraint.

Let Σ be a function from queries to constraints. We assume that in the domain of Σ no two queries have an equal predicate symbol, all queries have disjoint variables, and each query is mapped to a constraint whose free variables occur in the query. For example, consider $\Sigma = \{p(v_1) \mapsto (v_1 \geq 0), q(v_2, v_3) \mapsto (v_2 \leq f(v_3))\}$.

We use Σ function to transform the set of Horn clauses containing queries into a set of query-free clauses as follows. In each clause $s \in \mathcal{HC}$ we replace

each query q in s with the constraint $\Sigma(q')\sigma$ where q' is in the domain of Σ, queries q' and q have an equal predicate symbol, and $\sigma = match((q', q))$. For example, the above Σ transforms the clause $x \leq f(y) \wedge p(x) \wedge q(y, z) \rightarrow \mathit{false}$ into $x \leq f(y) \wedge (x \leq 0) \wedge (y \leq f(z)) \rightarrow \mathit{false}$.

Let $\mathcal{HC}_\Sigma$ be the set of query-free clauses obtained by applying Σ. Σ is a *solution* for $\mathcal{HC}$ if each clause $c_\Sigma$in $\mathcal{HC}_\Sigma$ is a valid implication, i.e, $\models c_\Sigma$, and the following condition holds for the uninterpreted function symbols occurring in the range of the solution function. An uninterpreted function symbol f can occur in the solution $\Sigma(q)$ for a query q if f appears in the atoms of a Horn clause from $\mathcal{HC}$ whose head depends on q and in the atoms of a Horn clause from $\mathcal{HC}$, whose head does not depend on q. For example, given the clauses $\{f(v_1) = 0 \rightarrow p(v_1), f(v_2) = 1 \rightarrow q(v_2), v_3 = v_4 \wedge p(v_3) \wedge q(v_4) \rightarrow \mathit{false}\}$ the function symbol f can appear in the solution of each query. A set of clauses is *satisfiable* if it has a solution.

4 Algorithm

Our goal is an algorithm for computing solutions for recursion-free Horn clauses over linear arithmetic, uninterpreted functions, and queries. This section presents our solving algorithm SolveHorn(LI+UIF).

See Figure 3. The algorithm SolveHorn(LI+UIF) consists of the following main steps. First, we compute a resolution tree R on the given set of Horn clauses. Next, we take a conjunction C of the leaves of the resolution tree and attempt to find a proof of its unsatisfiability. If no such proof can be found, then we report that there is no solution for the given set of Horn clauses. Otherwise, we proceed with the given proof by annotating its steps. Each intermediate atom derived by proof is annotated by a function that assigns constraints to nodes of the resolution tree. Finally, the annotation of the root of the proof yields a solution for the given set of Horn clauses.

In the rest of this section we provide a detailed presentation of the main steps of SolveHorn(LI+UIF).

4.1 Resolution Tree

We put together individual Horn clauses from $\mathcal{HC}$ by applying resolution inference. A *resolution tree* keeps the intermediate results of this computation. An edge of a *resolution tree* is a sequence of queries and atoms that is terminated by a query or *false*. Each edge consists of $n > 2$ elements. The first $n - 1$ elements represent the children nodes and the n-th element represents the parent node.

Given the set of Horn clauses $\mathcal{HC}$, we compute the corresponding resolution tree by applying the inference rules shown in Figure 4. Each rule takes as a premise a set of resolution trees and a Horn clause and infers an extended resolution tree.

The rule RInit initiates the resolution tree computation by inferring a tree from each clause that does not have any queries in its body. The atoms $a_1, \ldots, a_m$

algorithm SolveHorn(LI+UIF)
input
$\mathcal{HC}$: Horn clauses
vars
R : resolution tree
C : conjunctive constraint
P : proof tree
A : annotated proof tree
output
Σ : solution
begin
1 R := exhaustively apply RInit and RStep on $\mathcal{HC}$
2 $C := \bigwedge leaves(R)$
3 **if** exists P inferred from C by PHyp, PComb, and PCong
4 such that $\models (root(P) \rightarrow 1 \leq 0)$
5 **then**
6 A := exhaustively apply AHyp, AComb, and ACong on P
7 $false\ [\ \Pi\ ] := root(A)$
8 $\Sigma := \{(o, \pi) \mid (o, \pi) \in \Pi \wedge o \notin (leaves(R) \cup \{false\})\}$
9 **return** Σ
10 **else**
11 **return** "no solution exists"
end.

Fig. 3. Solving algorithm SolveHorn(LI+UIF). Line 7 extracts the partial solution Π annotating the root node of A. Line 8 obtains Σ by restricting the domain of Π to intermediate nodes of R, i.e., to the nodes that are labeled by queries.

$$\text{RInit}\ \frac{a_1 \wedge \cdots \wedge a_m \rightarrow h}{\{(a_1, \ldots, a_m, h)\}}$$

$$\text{RStep}\ \frac{\begin{array}{c} R_1 \quad \ldots \quad R_n \\ q_1 \wedge \cdots \wedge q_n \wedge a_1 \wedge \cdots \wedge a_m \rightarrow h \end{array}}{\begin{array}{l} R_1\sigma \cup \cdots \cup R_n\sigma \cup \\ \{(q_1, \ldots, q_n, a_1, \ldots, a_m, h)\}\sigma \end{array}}\ \sigma = (match(q_1, root(R_1)) + \cdots + match(q_n, root(R_n)))$$

Fig. 4. Resolution tree inference rules RInit and RStep

become the children of the node h. The rule RStep performs the extension of a set of trees computed so far using a Horn clause. The extension is only possible if the root nodes of the respective trees can be unified with the queries occurring in the body of the clause. This condition is formalized by the side condition requiring the existence of the most general unifier σ. The computed unifier is applied on the trees and the clause before they are combined into an extended resolution tree.

The resolution tree computation terminates since $\mathcal{HC}$ is recursion-free. Let R be the resulting tree. We consider the set of leaves of the tree, and take their conjunction $C = \bigwedge leaves(R)$.

$$\textsc{PHyp}\ \frac{}{t \leq 0}\ t \leq 0 \in \mathit{atoms}(C) \qquad \textsc{PComb}\ \frac{t_1 \leq 0 \quad \ldots \quad t_n \leq 0}{\lambda_1 t_1 + \cdots + \lambda_n t_n \leq 0}\ \lambda_1, \ldots, \lambda_n > 0$$

$$\textsc{PCong}\ \frac{\begin{array}{cc} t_1 - s_1 \leq 0 & s_1 - t_1 \leq 0 \\ \vdots & \vdots \\ t_n - s_n \leq 0 & s_n - t_n \leq 0 \end{array}}{f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n) \leq 0}\ f(t_1, \ldots, t_n), f(s_1, \ldots, s_n) \in \mathit{subterms}(C)$$

Fig. 5. Standard, complete proof rules PHyp, PComb, and PCong for combination of linear rational/real arithmetic and uninterpreted functions. C is the conjunction of leaves of the resolution tree R obtained from the Horn clauses $\mathcal{HC}$.

For a node o of the resolution tree, we define $\mathit{insym}(o)$ to be variables and uninterpreted function symbols that occur in atoms in the leaves of the subtree of o, and let $\mathit{outsym}(o)$ be variables and uninterpreted function symbols that occur in the leaves outside of the subtree of o. Formally, we have

$$\begin{aligned} \mathit{insym}(o) &= \bigcup\{\mathit{sym}(o') \mid o' \in \mathit{leaves}(\mathit{subtree}(o, R))\}\ , \\ \mathit{outsym}(o) &= \bigcup\{\mathit{sym}(o') \mid o' \in (\mathit{leaves}(R) \setminus \mathit{leaves}(\mathit{subtree}(o, R)))\}\ . \end{aligned}$$

The following proposition allows a transition from the clausal structure to the conjunction of atoms.

Proposition 1. *The set of Horn clauses $\mathcal{HC}$ is satisfiable if and only if the conjunction C is not satisfiable.*

The proof of Proposition 1 follows directly by applying induction over the resolution treee and relying on the definitions of RInit and RStep.

4.2 Proof Tree

The algorithm SolveHorn(LI+UIF) relies on unsatisfiability proofs. We use a standard set of proof rules for the combination of linear rational/real arithmetic and uninterpreted functions [17]. The implementation of the corresponding proof search procedure is irrelevant for our algorithm, yet we assume that this procedure is complete and use an existing tool for this task, e.g. [4, 7].

See Figure 5 for the proof rules, which we apply to the conjunction of atoms C. The rule PHyp states that atoms appearing in C are provable from C. The rule PComb infers that a set of inequalities implies a non-negatively weighted sum thereof. The congruence rule PCong represents a form of the functionality axiom, which states that equal inputs to a function lead to equal results. We are only interested in one inequality part of this axiom. The side condition of PCong is taken from the interpolating proof rules of [17], and simplifies the proof tree annotation in a way similar to [17].

We assume that there exists a mechanism that uniquely identifies the nodes of the proof tree, even in the presence of nodes that are labeled by equal inequalities,

$$\textsc{AHyp}\ \frac{}{t \le 0\ [\ \textsc{MkHyp}(t \le 0)\ ]}$$

$$\textsc{AComb}\ \frac{t_1 \le 0\ [\ \Pi_1\ ] \quad \ldots \quad t_n \le 0\ [\ \Pi_n\ ]}{\lambda_1 t_1 + \cdots + \lambda_n t_n \le 0\ [\ \textsc{MkComb}(\Pi_1, \ldots, \Pi_n, \lambda_1, \ldots, \lambda_n)\ ]}$$

$$\textsc{ACong}\ \frac{\begin{array}{cc} t_1 - s_1 \le 0\ [\ \Pi_1\ ] & s_1 - t_1 \le 0\ [\ \Pi'_1\ ] \\ \vdots & \vdots \\ t_n - s_n \le 0\ [\ \Pi_n\ ] & s_n - t_n \le 0\ [\ \Pi'_n\ ] \end{array}}{\begin{array}{r} f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n) \le 0\ [\ \textsc{MkCong}(f(t_1, \ldots, t_n), f(s_1, \ldots, s_n), \\ \Pi_1, \ldots, \Pi_n, \Pi'_1, \ldots, \Pi'_n)\ ] \end{array}}$$

Fig. 6. Annotation rules. The function MkHyp, MkComb, and MkCong are shown in Figure 7.

for example by numbering them. For clarity of exposition, we omit any details of such mechanism and assume that the node label carries all necessary information.

If no proof can be found then our algorithm reports that no solution exists. Otherwise, let P be the discovered proof. We assume that P is represented by a tree where nodes are atoms and the children of a node are defined by the rules PHyp, PComb, and PCong. Furthermore, we assume that each edge is labeled by the name of the proof rule that created it.

4.3 Annotated Proof Tree

We construct a solution for the given Horn clauses through an iterative process, where the intermediate results are called *partial solutions*. Each partial solution is parameterized by a constraint c. A c-partial solution Π for the resolution tree R is a function from nodes of the resolution tree, $nodes(R)$, to constraints that satisfies the following conditions.

$$(\forall o \in leaves(R) :\ (\models o \rightarrow \Pi(o))) \wedge \tag{PS1}$$

$$(\forall (o^1, \ldots, o^m, o) \in R : \models \Pi(o^1) \wedge \cdots \wedge \Pi(o^m) \rightarrow \Pi(o)) \wedge \tag{PS2}$$

$$(\models \Pi(false) \rightarrow c) \wedge \tag{PS3}$$

$$(\forall o \in nodes(R) :\ sym(\Pi(o)) \subseteq (insym(o) \cap outsym(o)) \cup sym(c)) \tag{PS4}$$

Our annotation uses constraints of the following form, called *solution constraints*.

$$\text{solution constraints} \ni \pi ::= t \le 0 \mid \hat{c} \wedge (\hat{c} \rightarrow \pi)$$

To simplify the presentation, we represent a solution constraint

$$C_1 \wedge (D_1 \rightarrow (\ldots C_r \wedge (D_r \rightarrow p \le 0)))$$

as a pair consisting of a corresponding sequence and a term $\langle(\,(C_1, D_1), \ldots, (C_r, D_r)\,), p\rangle$. A solution constraint $p \leq 0$ is represented by $\langle[], p\rangle$.

Given the proof tree P, we annotate its nodes with partial solutions using the rules shown in Figure 6 and auxiliary functions shown in Figure 7. The rule AHyp annotates each leaf of the proof tree with the result of applying the function MkHyp. The annotation is enclosed by a pair of square brackets. The rule AComb shows how to annotate a parent node when provided with an annotation of its children in case when the parent was obtained by a non-negatively weighted sum. The parent annotation is computed by MkComb. Similarly, the rule ACong annotates parent nodes obtained by the congruence rule.

For each node of R at line 6, ACong has four cases that deal with the difficulty of solving Horn clauses over uninterpreted functions, i.e., a sub term may contain variables that are not allowed to appear in the partial solutions. The proof of theorem 3 explains how these cases avoid such variables in the partial solutions.

We annotate P and obtain an annotated proof tree A. Our algorithm SolveHorn(LI+UIF) uses the annotation of the root of A to derive a solution to the Horn clauses $\mathcal{HC}$.

5 Correctness and Complexity

This section presents the correctness and complexity properties of our algorithm. The corresponding proofs are in Appendix C of extended version of this paper [10].

The correctness of our algorithm follows from Proposition 1 and Theorems 1-3 below. First, we establish that a $(1 \leq 0)$-partial solution, which satisfies Equations (PS1)–(PS4), defines a solution for the given Horn clauses.

Theorem 1. *$(1 \leq 0)$-partial solution defines a solution of the Horn clauses.*

Now, we show that the annotations computed by the rules in Figure 6 satisfy the partial solution conditions in Equations (PS1)–(PS4). This step relies on the following inductive invariant.

Definition 1 ($t \leq 0$-annotation invariant). *Π is $t \leq 0$-annotation invariant for the resolution tree R if there exists $r \geq 0$ such that for each $o \in nodes(R)$ the following conditions hold.*

- *$\Pi(o)$ is a solution constraint such that*

$$\Pi(o) = \langle((C_1, D_1), \ldots, (C_r, D_r)), p\rangle. \tag{AI-1}$$

- *If $o \in leaves(R)$ then*

$$\left(\forall i \in 1..r : \quad \models o \wedge \bigwedge_{k=1}^{i-1} D_k \rightarrow C_i\right) \wedge \tag{AI-2a}$$

$$\left(\models o \wedge \bigwedge_{k=1}^{r} D_k \rightarrow p \leq 0\right). \tag{AI-2b}$$

function MkHyp
input
$t \leq 0$: inequality term/node in R
begin
1 **for** each $o \in nodes(R)$ **do**
2 **if** $t \leq 0 \in leaves(subtree(o, R))$ **then**
3 $\Pi(o) := \langle [], t \rangle$
4 **else**
5 $\Pi(o) := \langle [], 0 \rangle$
6 **return** Π
end

function MkComb
input
$\Pi_1, \ldots, \Pi_n$: partial solutions
$\lambda_1, \ldots, \lambda_n$: constants
begin
1 **for** each $o \in nodes(R)$ **do**
2 **for** each $i \in 1..n$ **do**
3 $\langle L_i, t_i \rangle := \Pi_i(o)$
4 $L := L_1 \bullet \cdots \bullet L_n$
5 $t := \lambda_1 t_1 + \cdots + \lambda_n t_n$
6 $\Pi(o) := \langle L, t \rangle$
7 **return** Π
end

function MkCong
input
$f(t_1, \ldots, t_n)$, $f(s_1, \ldots, s_n)$: terms
$\Pi_1, \ldots, \Pi_n$, $\Pi'_1, \ldots, \Pi'_n$: partial solutions
begin
1 **for** each $o \in nodes(R)$ **do**
2 **for** each $i \in 1..n$ **do**
3 $\langle L_i, p_i \rangle := \Pi_i(o)$
4 $\langle L'_i, p'_i \rangle := \Pi'_i(o)$
5 $(C, D, p) :=$
6 **match** $sym(f(t_1, \ldots, t_n)) \subseteq outsym(o)$,
7 $sym(f(s_1, \ldots, s_n)) \subseteq outsym(o)$ **with**
8 | $true, true$ -> $(\bigwedge_{i=1}^{n}(p_i \leq 0 \wedge p'_i \leq 0), true, 0)$
9 | $true, false$ -> $(\bigwedge_{i=1}^{n} p_i + p'_i \leq 0, \bigwedge_{i=1}^{n} -p_i - p'_i \leq 0,$
10 $f(s_1 + p_1, \ldots, s_n + p_n) - f(s_1, \ldots, s_n))$
11 | $false, true$ -> $(\bigwedge_{i=1}^{n} p_i + p'_i \leq 0, \bigwedge_{i=1}^{n} -p_i - p'_i \leq 0,$
12 $f(t_1, \ldots, t_n) - f(t_1 + p'_1, \ldots, t_n + p'_n))$
13 | $false, false$ -> $(true, \bigwedge_{i=1}^{n}(t_i - s_i - p_i \leq 0 \wedge s_i - t_i - p'_i \leq 0),$
14 $f(t_1, \ldots, t_n) - f(s_1, \ldots, s_n))$
15 $\Pi(o) := \langle L_1 \bullet \cdots \bullet L_n \bullet L'_1 \bullet \cdots \bullet L'_n \bullet (C, D), p \rangle$
16 **return** Π
end

Fig. 7. Computation of partial solutions to annotate nodes of the proof tree, as shown in Figure 6. We use $\bullet$ to denote concatenation of sequences.

– *If* $(o^1, \ldots, o^m, o) \in R$ *and* $\forall j \in 1..m : \Pi(o^j) = \langle((C_1^j, D_1^j), \ldots, (C_r^j, D_r^j)), p^j\rangle$ *then*

$$\left(\forall i \in 1..r : \quad \models \left(\bigwedge_{k=1}^{i}\bigwedge_{l=1}^{m} C_k^l\right) \ \wedge \ \bigwedge_{k=1}^{i-1} D_k \rightarrow C_i\right) \wedge \tag{AI-3a}$$

$$\left(\begin{array}{l}\forall i \in 1..r \\ \forall j \in 1..m\end{array} : \quad \models \begin{array}{l}\left(\bigwedge_{l \in 1..m \setminus \{j\}} C_i^l\right) \wedge \\ \left(\bigwedge_{k=1}^{i-1}\bigwedge_{l=1}^{m} C_k^l\right) \wedge \bigwedge_{k=1}^{i} D_k \rightarrow D_i^j\end{array}\right) \wedge \tag{AI-3b}$$

$$\left(\models \left(\bigwedge_{k=1}^{r}\bigwedge_{l=1}^{m} C_k^l\right) \ \wedge \ \bigwedge_{k=1}^{r} D_k \rightarrow p - p^1 - \cdots - p^m \leq 0\right). \tag{AI-3c}$$

– *If* $o = \mathit{false}$ *then*

$$p = t \wedge \forall i \in 1..r : D_i = C_i = \mathit{true}. \tag{AI-4}$$

– *Conditions on symbol appearance:*

$$\mathit{sym}(\{C_1, \ldots, C_r, D_1, \ldots, D_r, p \leq 0\}) \subseteq \mathit{insym}(o) \wedge \tag{AI-5}$$

$$\mathit{sym}(\{C_1, \ldots, C_r, D_1, \ldots, D_r, t - p \leq 0\}) \subseteq \mathit{outsym}(o). \tag{AI-6}$$

The above definition act as an intermediate step. In theorem 2, we show that a $t \leq 0$-annotation invariant satisfies all the conditions for being a $t \leq 0$-partial solution.

Theorem 2. *Each* $t \leq 0$*-annotation invariant is a* $t \leq 0$*-partial solution.*

Now, we show that the presented algorithm computes the partial solutions that satisfies the invariant.

Theorem 3. *The annotation rules in Figure 6 compute annotation invariants.*

Theorem 4 (Complexity). *The application of the annotation rules from Figure 6 takes time proportional to the product of the size of the proof tree and the size of the resolution tree. The size of the resolution tree is linear in the size of the corresponding set of recursion-free, tree-like Horn clauses.*

Note that we present the complexity of our algorithm in terms of the size of the proof tree. Since the size of a resolution tree can also be exponential in the size of the set of Horn clauses, the size of a proof tree can be exponential.

6 Conclusion

We presented an algorithm for computing solutions for recursion-free Horn clauses over the combination of linear rational/real arithmetic, uninterpreted functions, and queries.

Connection to Interpolation. The interpolation algorithm presented in [17] is a special case for the algorithm presented in this paper. As illustrated in the introduction, an interpolation problem can be reduced to solving a set of recursion-free Horn clauses. The set of Horn clauses resulting from an interpolation problem has only one unknown query. Therefore, the corresponding resolution tree obtained from the set of Horn clauses contains only one internal node. The partial solution of this internal node in the $(1 \leq 0)$-partial solution will be the interpolant. In this special case, we only need to track partial solutions of the internal node in the annotated proof tree. We can transform our algorithm for this case such that nodes of the proof tree are annotated with a formula corresponding to the partial solution of this internal node. The resulting algorithm will be the algorithm presented in [17].

Our algorithm can be directly applied to support abstraction and refinement tasks for the verification of programs with procedures, threads and higher order functions.

Acknowledgment. Ashutosh Gupta was supported in part by the DFG Graduiertenkolleg 1480 (PUMA), FWF NFN Grant No S11407-N23 (RiSE), and the ERC Advanced Grant QUAREM.

References

1. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD (2009)
2. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
3. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
4. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
5. Cimatti, A., Griggio, A., Sebastiani, R.: Interpolant Generation for UTVPI. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 167–182. Springer, Heidelberg (2009)
6. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Logic 12 (November 2010)
7. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Goel, A., Krstić, S., Tinelli, C.: Ground Interpolation for Combined Theories. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 183–198. Springer, Heidelberg (2009)
9. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL (2011)
10. Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free Horn clauses over LI+UIF (2011), http://pub.ist.ac.at/~agupta/papers/HornLIUIF.pdf

11. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL (2010)
12. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
13. Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig interpolation for linear Diophantine (dis)equations and linear modular equations. In: Formal Methods in System Design, pp. 6–39 (2009)
14. Jhala, R., Majumdar, R.: Counterexample refinement for functional programs (2009), http://www.cs.ucla.edu/~rupak/Papers/CEGARFunctional.ps
15. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
16. Kroening, D., Leroux, J., Rümmer, P.: Interpolating Quantifier-Free Presburger Arithmetic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 489–503. Springer, Heidelberg (2010)
17. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
18. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
19. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
20. Terauchi, T.: Dependent types from counterexamples. In: POPL (2010)
21. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: PPDP (2009)
22. Yorsh, G., Musuvathi, M.: A Combination Method for Generating Interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Macro Tree Transformations of Linear Size Increase Achieve Cost-Optimal Parallelism

Akimasa Morihata

Research Institute of Electrical Communication, Tohoku University
morihata@riec.tohoku.ac.jp

Abstract. This paper studies parallel evaluation of tree transformations, in particular accumulative ones. Accumulation is a ubiquitous programming pattern. However, since accumulation usually imposes restrictions on evaluation orders, accumulative tree transformations appear to be unsuitable for parallel evaluation. We propose a parallel evaluation method for a large class of tree-to-tree recursive functions, which may contain accumulations, higher-order terms, and function compositions. Our parallel evaluation method achieves optimal parallel speedup if the transformation is of linear size increase, namely, the size of each output is linearly bounded by the size of the corresponding input. Our result is based on the theory of macro tree transducers and that of parallel tree contractions. The main contribution is to reveal a good collaboration between them.

1 Introduction

Recent popularization of parallel computers has introduced an additional difficulty to programming. Now we cannot achieve good performance without writing parallel programs. However, there are a lot of programming patterns that appear to be not suitable for parallel evaluation. One such is accumulation. As an example, consider the following accumulative function, *flatten*, which gathers values in a tree.

$$\begin{aligned} &\mathit{flatten}\ t = \mathit{flat}\ t\ [] \\ &\mathit{flat}\ (\mathsf{Tip}\ v)\ \mathit{acc} &&= v : \mathit{acc} \\ &\mathit{flat}\ (\mathsf{Bin}(l, v, r))\ \mathit{acc} &&= \mathit{flat}\ l\ (v : \mathit{flat}\ r\ \mathit{acc}) \end{aligned}$$

It seems impossible to evaluate *flatten* in parallel. Auxiliary function *flat* forces right-to-left traversals and prohibits parallel processing. Accumulation usually imposes restrictions on evaluation orders and thereby makes parallel evaluation difficult. Yet, accumulation is a really ubiquitous programming pattern. We would like to discover and exploit parallelism from accumulative programs.

In fact, *flatten* can be evaluated in parallel, because it is equivalent to the following $\mathit{flatten}_{\mathrm{para}}$, in which $+\!\!+$ denotes the list concatenation and we regard it as an O(1) operation.

$$\begin{aligned} &\mathit{flatten}_{\mathrm{para}}\ (\mathsf{Tip}\ v) &&= [v] \\ &\mathit{flatten}_{\mathrm{para}}\ (\mathsf{Bin}(l, v, r)) &&= \mathit{flatten}_{\mathrm{para}}\ l +\!\!+ [v] +\!\!+ \mathit{flatten}_{\mathrm{para}}\ r \end{aligned}$$

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 204–219, 2011.

Since $\mathit{flatten}_{\mathrm{para}}$ does not use any accumulative parameters, we can process independent subtrees in parallel.

We have seen that accumulative programs are usually hard to evaluate in parallel but some of them may possess hidden parallelism. It is natural to wonder what class of accumulative programs does. The subject of this paper is to answer this question—in fact, we answer a more generic question that is not restricted to accumulations. Informally speaking, we will show the following.

> *A class of tree-to-tree transformations, which may contain accumulations, higher-order terms, and function compositions, can be efficiently evaluated in parallel if they do not output a tree significantly larger than its corresponding input.*

To model possibly accumulative tree transformations, we employ *macro tree transducers* [1,2], abbreviated to MTTs. Roughly speaking, MTTs are first-order structural recursive functions that concern only constructors.

The notion of "efficient" is ambiguous. We seek for tree transformations that can be evaluated in time $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$, where P and N respectively denote the number of processors and the size of the input tree. It is *cost optimal* for $\mathrm{P} \in \mathrm{O}(\mathrm{N}/\log \mathrm{N})$: in comparison to $\mathrm{O}(\mathrm{N})$ sequential evaluation, the parallel evaluation, which takes $\mathrm{O}(\mathrm{N}/\mathrm{P})$ time, shows asymptotically linear parallel speedup. Cost optimal is difficult to achieve. For instance, the naive divide-and-conquer parallel evaluation of $\mathit{flatten}_{\mathrm{para}}$ is not sufficient because of its poor parallel speedup for monadic, list-like input trees. We make use of *parallel tree contraction algorithms* [3] to achieve cost-optimal parallelism.

First, we prove that two variants of MTTs, top-down relabelings with regular lookaheads and strongly single-use restricted MTTs, can be efficiently evaluated based on parallel tree contraction algorithms (Section 4). Then, we consider tree transformations of *linear size increase*, namely those for which the size of each output is linearly bounded by the size of the corresponding input. Our parallel evaluation method can deal with a lot of tree transformations of linear size increase by transforming them to a composition of a strongly single-use restricted MTT and a top-down relabeling with regular lookaheads (Section 5).

It should be noted that our result is rather straightforward from the theory of MTTs and that of parallel tree contraction algorithms. Our main contribution is a discovery of a good collaboration between them, which would open a new vista of parallel programming.

2 Preliminary

2.1 Trees

Ranked alphabet Σ is a finite set of ranked symbols. $\Sigma^{(k)}$ denotes the set of k-ranked symbols in Σ. Given ranked alphabet Σ and finite set of variables X, $T_\Sigma(X)$ denotes the set of trees that are constructed from Σ and X in the standard manner. We abbreviate $T_\Sigma(\emptyset)$ to T_Σ. We may call a tree a context if it

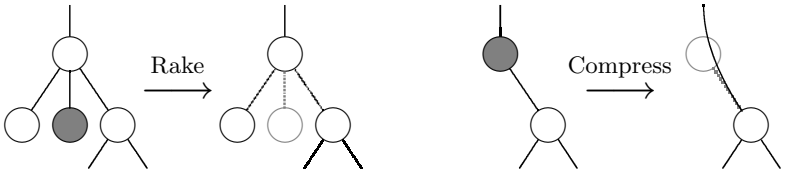


Fig. 1. Primitive contraction operations applied to the gray nodes

contains variables. We denote $|t|$ to mean the size of tree t, namely the number of symbols in t. Given $t_1, t_2 \in T_\Sigma(X)$ and $x \in X$, $t_1[x \mapsto t_2] \in T_\Sigma(X)$ denotes a tree obtained by substituting all occurrences of x in t_1 by t_2. We may write $t_1[x_i \mapsto t_i]_{i \in I}$ to denote a series of substitutions indexed by I. We assume that $t_1[x \mapsto t_2]$ can be calculated in time $\mathrm{O}(1)$ if x appears at most once in t_1. This can be easily achieved by preparing a pointer for each variable occurrence.

2.2 Parallel Tree Contraction

Our computation model is EREW PRAM (Exclusive-Read Exclusive-Write Parallel Random Access Machines), in which each processor can read/write each memory address in $\mathrm{O}(1)$ time if no other ones simultaneously read/write there. P and N respectively denote the number of processors and the size of the input.

Parallel tree contraction algorithms [3] collapse a tree through simultaneous applications of primitive contraction operations called *Rake* and *Compress*. As depicted in Fig. 1, a Rake operation removes a leaf, and a Compress operation removes an internal node that has no sibling.

Several efficient parallel tree contraction algorithms are known. Here we just state the result.

Theorem 1 ([3]). *A tree of* N *nodes can be collapsed to a leaf by Rake and Compress operations in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$. □

Parallel tree contraction can implement several tree operations. For example, consider a parallel evaluation of an expression that consists of integers and $+$ and $\times$ operators. We can regard each internal node as a closure whose operands are missing. From this viewpoint, Rake is a (possibly partial) application and Compress is a function composition. In this case, parallel tree contraction provides efficient expression evaluation, because partial applications will result in closures of the form $(\lambda x.\ a \times x + b)$, where a and b are some integers, and their compositions do not result in larger closures: $(\lambda x.\ a_1 \times x + b_1) \circ (\lambda x.\ a_2 \times x + b_2) = (\lambda x.\ (a_1 \times a_2) \times x + (a_1 \times b_2 + b_1))$.

We generalize and formalize the idea described above. We consider evaluation of a tree-structured circuit. Let A^k be the set of all k-tuples of A elements.

A bottom-up circuit is a triple (t, h, A): $t \in T_\Sigma$ is the input tree and gives an underlying structure; h associates each t's node $\sigma \in \Sigma^{(0)}$ with initial input value $h_\sigma \in A$ and $\sigma \in \Sigma^{(k)}$ $(k \geq 1)$ with gate $h_\sigma : A^k \to A$ that takes its inputs from its children and sends the calculated value to its parent. The expression evaluation

considered above is an example of a bottom-up circuit, which consists of adder and multiplier gates.

A top-down circuit is a quadruple (t, h, A, a_0): $t \in T_\Sigma$ is the input tree; h associates each t's node $\sigma \in \Sigma^{(k)}$ $(k \geq 1)$ with gate $h_\sigma : A \to A^k$ that takes its inputs from its parent and sends the calculated values to its children; $a_0 \in A$ is the initial input given to the root; $\sigma \in \Sigma^{(0)}$ corresponds to an output gate.

It is known that both bottom-up and top-down circuits can be implemented by parallel tree contraction under certain requirements.

Theorem 2 ([4,3]). *The output of the root of a bottom-up circuit (t, h, A) can be calculated in time $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$, if there exists a set of functions, say $\mathcal{F}$, that satisfies the following conditions.*

- *For every $1 \leq i \leq k$, $\sigma \in \Sigma^{(k)}$, and $\{a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k\} \subseteq A$, function f defined by $f(x) = h_\sigma(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_k)$ is in $\mathcal{F}$.*
- *Every element of $\mathcal{F}$ can be evaluated in constant time.*
- *For every $f_1, f_2 \in \mathcal{F}$, we can calculate $f_3 \in \mathcal{F}$ in constant time such that $f_1 \circ f_2 = f_3$.* □

Theorem 3 ([4,5]). *All the outputs of the gates of a bottom-up circuit (t, h, A) can be calculated in time $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$, if we can duplicate each element of A in constant time and the premise of Theorem 2 holds.* □

Theorem 4 ([4,5]). *All the outputs of the gates of a top-down circuit (t, h, A, a) can be calculated in time $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$, if we can duplicate each element of A in constant time and there exists a set of functions, say $\mathcal{F}$, that satisfies the following conditions.*

- *Let π_i be the function that extract the i-th element from a tuple. For every $1 \leq i \leq k$ and $\sigma \in \Sigma^{(k)}$, $\pi_i \circ h_\sigma \in \mathcal{F}$.*
- *Every element of $\mathcal{F}$ can be evaluated in constant time.*
- *For every $f_1, f_2 \in \mathcal{F}$, we can calculate $f_3 \in \mathcal{F}$ in constant time such that $f_1 \circ f_2 = f_3$.* □

Roughly speaking, the premises require that a series of Rake operations eventually results in one in $\mathcal{F}$ and their compositions and applications can be performed efficiently. Because we duplicate calculated values for calculating all the outputs of the gates, efficient duplication is additionally required in Theorems 3 and 4.

3 Tree Transducers

3.1 Macro Tree Transducers

We study parallel evaluation of *macro tree transducers* [1,2].

Definition 1. *A (total deterministic)* macro tree transducer *(abbreviated to MTT) is tuple $(Q, \Sigma, \Delta, q_0, R)$: Q, Σ, and Δ are ranked alphabets, which respectively characterize the recursive functions, the input, and the output; $q_0 \in Q^{(1)}$ is*

*the initial function; R is a set of equations. For each $q \in Q^{(m+1)}$ and $\sigma \in \Sigma^{(k)}$, R contains exactly one rule, called the $\langle q, \sigma \rangle$-*rule*, of the following form.*

$$q(\sigma(x_1, \cdots, x_k), y_1, \ldots, y_m) = expr$$

The syntax of expr is defined as follows, where $1 \leq i \leq k$, $1 \leq j \leq m$, $p \in Q^{(n+1)}$, and $\delta \in \Delta^{(l)}$.

$$expr ::= p(x_i, expr_1, \ldots, expr_n) \mid y_j \mid \delta(expr_1, \ldots, expr_l)$$

The class of tree transformations realized by MTTs is denoted by MTT. □

We regard MTT $M = (Q, \Sigma, \Delta, q_0, R)$ as a functional program. Given input $t \in T_\Sigma$, the output of M, denoted by $M(t)$, is a tree in T_Δ calculated by reducing $q_0(t)$ based on standard reduction semantics. Since we deal with total, deterministic, and terminating MTTs, the choice of reduction strategies is not important. We may omit rules that are never used.

From the perspective of functional programs, MTTs have two major restrictions. First, we cannot use function parameters. Second, an MTT should distinguish the input and the output and cannot traverse or destruct more than one tree. In particular, an MTT cannot traverse trees that are outputted from functions. We will loosen these restrictions in Section 5.

Examples. The *flatten* seen in the introduction can be described as an MTT.

$$Flatten = (\{flatten, flat\}, \{\mathsf{Bin}, \mathsf{Tip}\}, \{\mathsf{Cons}, \mathsf{Nil}\}, flatten, R)$$
$$R = \left\{ \begin{array}{ll} flatten(\mathsf{Tip}(v)) & = \mathsf{Cons}(v, \mathsf{Nil}), \\ flatten(\mathsf{Bin}(x_1, v, x_2)) & = flat(x_1, \mathsf{Cons}(v, flat(x_2, \mathsf{Nil}))), \\ flat(\mathsf{Tip}(v), y) & = \mathsf{Cons}(v, y), \\ flat(\mathsf{Bin}(x_1, v, x_2), y) & = flat(x_1, \mathsf{Cons}(v, flat(x_2, y))) \end{array} \right\}$$

Note that, for simplicity, this program slightly violates the definition of MTTs: The variable v cannot be directly used in right-hand side expressions. This is not harmful for our objective. It is sufficient to regard v as a 0-ary constructor.

Another example is the following *Exp* that yields 2^n for given n.

$$Exp = (\{exp_0, exp\}, \{\mathsf{S}, \mathsf{Z}\}, \{\mathsf{S}, \mathsf{Z}\}, exp_0, R)$$
$$R = \left\{ \begin{array}{ll} exp_0(\mathsf{Z}) & = \mathsf{S}(\mathsf{Z}), \\ exp_0(\mathsf{S}(n)) & = exp(n, exp(n, \mathsf{Z})), \end{array} \quad \begin{array}{ll} exp(\mathsf{Z}, y) & = \mathsf{S}(y), \\ exp(\mathsf{S}(n), y) & = exp(n, exp(n, y)) \end{array} \right\}$$

The third example, *A2B*, extracts the longest consecutive subsequence that starts from A and ends with B.

$$A2B = (\{f_0, f_\mathsf{A}, has_\mathsf{B}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{N}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{N}\}, f_0, R)$$
$$R = \left\{ \begin{array}{ll} f_0(\mathsf{N}) & = \mathsf{N}, \\ f_0(\mathsf{A}(x)) & = has_\mathsf{B}(x, \mathsf{N}, \mathsf{A}(f_\mathsf{A}(x))), \\ f_0(\mathsf{B}(x)) & = f_0(x), \\ has_\mathsf{B}(\mathsf{N}, y_1, y_2) & = y_1, \\ has_\mathsf{B}(\mathsf{A}(x), y_1, y_2) & = has_\mathsf{B}(x, y_1, y_2), \\ has_\mathsf{B}(\mathsf{B}(x), y_1, y_2) & = y_2 \end{array} \quad \begin{array}{ll} f_\mathsf{A}(\mathsf{N}) & = \mathsf{N}, \\ f_\mathsf{A}(\mathsf{A}(x)) & = \mathsf{A}(f_\mathsf{A}(x)), \\ f_\mathsf{A}(\mathsf{B}(x)) & = has_\mathsf{B}(x, \mathsf{B}(\mathsf{N}), \mathsf{B}(f_\mathsf{A}(x))), \end{array} \right\}$$

Function has_B is important. It takes two additional parameters, and returns the first if the sequence does not contain B, and the second otherwise.

3.2 Strongly Single-Use Restricted MTTs

Parallel evaluation of MTTs seems difficult in general, and therefore, we consider its subclass, namely *strongly single-use restricted* MTTs [6], which use arguments and results of functions at most once.

Definition 2. *MTT* $(Q, \Sigma, \Delta, q_0, R)$ *is said to be* single-use restricted in the parameters *if for every* $\langle q, \sigma\rangle$*-rule* $(q \in Q^{(m+1)})$ *in* R*, its right-hand side contains at most one occurrence of each* y_i $(1 \leq i \leq m)$*.*

Definition 3. *MTT* $(Q, \Sigma, \Delta, q_0, R)$ *is said to be* strongly single-use restricted in the input *if for every* $p \in Q^{(m+1)}$*,* $\sigma \in \Sigma^{(k)}$*, and* x_j $(1 \leq j \leq k)$*, the* $\langle q, \sigma\rangle$*-rules in* R *contain at most one occurrence of the form* $p(x_j, expr_1, \ldots, expr_m)$*.*

Definition 4. *An MTT is said to be* strongly single-use restricted *(abbreviated to SSUR) if it is both single-use restricted in the parameters and strongly single-use restricted in the input. The corresponding class of tree transformations is denoted by* MTT_{ssur}*.* □

3.3 Top-Down Relabeling with Regular Lookahead

Top-down relabeling [6] is another class of tree transformations. We use a variant that has regular lookaheads. Lookaheads, specified by tree automata, require subtrees to be in a certain shape. We refer readers who are not familiar with tree automata to the text [7]. For deterministic bottom-up tree automaton $\mathcal{A}$, we write $s \in \mathcal{A}$ to mean that s is a state of $\mathcal{A}$, and write $s = \mathcal{A}(t)$ to mean that the run of $\mathcal{A}$ for tree t yields state $s \in \mathcal{A}$.

Definition 5. *A (total deterministic)* top-down relabeling with regular lookaheads *(abbreviated to* T^{R}*-REL) is tuple* $(Q, \Sigma, \Delta, q_0, R, \mathcal{A})$*: Q is a finite set of recursive function names; Σ and Δ are ranked alphabets, which respectively characterize the input and the output; $q_0 \in Q$ is the initial function; $\mathcal{A}$ is a deterministic bottom-up tree automaton. R is a set of equations. For each $q \in Q$, $\sigma \in \Sigma^{(k)}$, and $s_i \in \mathcal{A}$ $(1 \leq i \leq k)$, R contains exactly one rule of the following form, where $\delta \in \Delta$ has the same rank as σ.*

$$q(\sigma(x_1, \ldots, x_k))\ \langle s_1, \ldots, s_k\rangle = \delta(q_1(x_1), \ldots, q_k(x_k))$$

The class of tree transformations realized by them is denoted by T^{R}-REL. □

T^{R}-RELs rename some constructors. The lookaheads are analogous to guards. Given rule $q(\sigma(x_1, \cdots, x_k))\ \langle s_1, \ldots, s_k\rangle = \xi$, $q(\sigma(t_1, \ldots, t_k))$ is reduced to $\xi[x_i \mapsto t_i]_{1 \leq i \leq k}$ if and only if $\mathcal{A}(t_i) = s_i$ holds for all $1 \leq i \leq k$. We may omit lookaheads if it is not necessary.

Examples (Continued). *Flatten* is not SSUR. Both of the $\langle$*flatten*, Bin$\rangle$-rule and the $\langle$*flat*, Bin$\rangle$-rule contain a function call of *flat* for x_1, and thus, it violates the strongly single-use restriction in the inputs. Yet, it can be decomposed to a T^{R}-REL and an SSUR MTT, $Flatten(t) = Flatten_{\text{ssur}}(Flatten_{\text{REL}}(t))$, defined as follows.

$$Flatten_{\mathrm{REL}} = (\{rel_0, rel\}, \{\mathsf{Bin}, \mathsf{Tip}\}, \{\mathsf{Bin}, \mathsf{Bin}_0, \mathsf{Tip}\}, rel_0, R)$$
$$R = \left\{ \begin{array}{ll} rel_0(\mathsf{Tip}(v)) & = \mathsf{Tip}(v), \\ rel_0(\mathsf{Bin}(x_1, v, x_2)) & = \mathsf{Bin}_0(rel(x_1), v, rel(x_2)), \\ rel(\mathsf{Tip}(v)) & = \mathsf{Tip}(v), \\ rel(\mathsf{Bin}(x_1, v, x_2)) & = \mathsf{Bin}(rel(x_1), v, rel(x_2)) \end{array} \right\}$$

$$Flatten_{\mathrm{ssur}} = (\{flatten, flat\}, \{\mathsf{Bin}_0, \mathsf{Bin}, \mathsf{Tip}\}, \{\mathsf{Cons}, \mathsf{Nil}\}, flatten, R)$$
$$R = \left\{ \begin{array}{ll} flatten(\mathsf{Tip}(v)) & = \mathsf{Cons}(v, \mathsf{Nil}), \\ flatten(\mathsf{Bin}_0(x_1, v, x_2)) & = flat(x_1, \mathsf{Cons}(v, flat(x_2, \mathsf{Nil}))), \\ flat(\mathsf{Tip}(v), y) & = \mathsf{Cons}(v, y), \\ flat(\mathsf{Bin}(x_1, v, x_2), y) & = flat(x_1, \mathsf{Cons}(v, flat(x_2, y))) \end{array} \right\}$$

$Flatten_{\mathrm{REL}}$ renames the root constructor to Bin_0. This enables us to distinguish the node processed by *flatten* (therefore, we omit the $\langle flatten, \mathsf{Bin}\rangle$-rule and the $\langle flat, \mathsf{Bin}_0\rangle$-rule, which are unnecessary) and thereby makes $Flatten_{\mathrm{ssur}}$ SSUR.

Similarly, we can decompose $A2B$ as $A2B(t) = A2B_{\mathrm{ssur}}(A2B_{\mathrm{REL}}(t))$. We use tree automaton $\mathcal{A}$ such that $\mathcal{A}(t) = s_1$ if t contains B and $\mathcal{A}(t) = s_2$ otherwise.

$$A2B_{\mathrm{REL}} = (\{rel_0, rel_{\mathsf{A}}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{N}\}, \{\mathsf{A}, \mathsf{A}', \mathsf{B}, \mathsf{B}', \mathsf{N}\}, rel_0, R, \mathcal{A})$$
$$R = \left\{ \begin{array}{llll} rel_0(\mathsf{N}) & = \mathsf{N}, & rel_{\mathsf{A}}(\mathsf{N}) & = \mathsf{N}, \\ rel_0(\mathsf{A}(x))\ \langle s_1\rangle & = \mathsf{A}(rel_{\mathsf{A}}(x)), & rel_{\mathsf{A}}(\mathsf{A}(x)) & = \mathsf{A}(rel_{\mathsf{A}}(x)), \\ rel_0(\mathsf{A}(x))\ \langle s_2\rangle & = \mathsf{A}'(rel_0(x)), & rel_{\mathsf{A}}(\mathsf{B}(x))\ \langle s_1\rangle & = \mathsf{B}(rel_{\mathsf{A}}(x)), \\ rel_0(\mathsf{B}(x)) & = \mathsf{B}(rel_0(x)), & rel_{\mathsf{A}}(\mathsf{B}(x))\ \langle s_2\rangle & = \mathsf{B}'(rel_{\mathsf{A}}(x)) \end{array} \right\}$$

$$A2B_{\mathrm{ssur}} = (\{f_0, f_{\mathsf{A}}\}, \{\mathsf{A}, \mathsf{A}', \mathsf{B}, \mathsf{B}', \mathsf{N}\}, \{\mathsf{A}, \mathsf{B}, \mathsf{N}\}, f_0, R)$$
$$R = \left\{ \begin{array}{llll} f_0(\mathsf{N}) & = \mathsf{N}, & f_{\mathsf{A}}(\mathsf{N}) & = \mathsf{N}, \\ f_0(\mathsf{A}(x)) & = \mathsf{A}(f_{\mathsf{A}}(x)), & f_{\mathsf{A}}(\mathsf{A}(x)) & = \mathsf{A}(f_{\mathsf{A}}(x)), \\ f_0(\mathsf{A}'(x)) & = \mathsf{N}, & f_{\mathsf{A}}(\mathsf{B}(x)) & = \mathsf{B}(f_{\mathsf{A}}(x)), \\ f_0(\mathsf{B}(x)) & = f_0(x), & f_{\mathsf{A}}(\mathsf{B}'(x)) & = \mathsf{B}(\mathsf{N}) \end{array} \right\}$$

$A2B_{\mathrm{REL}}$ marks constructors whose subtrees do not contain B. Then, $A2B_{\mathrm{ssur}}$ can avoid performing additional traversals like has_{B}.

Exp is not SSUR; moreover, we cannot express *Exp* by any compositions of T^{R}-RELs and SSUR MTTs. This can be proved by the height property of T^{R}-RELs and SSUR MTTs [6].

4 Parallel Evaluation of Tree Transformations

Apart from accumulative parameters, there are two major difficulties in parallel evaluation of MTTs. First, it is in general difficult to predict how an MTT performs recursive calls. For example, *Exp* causes 2^k recursive calls for the subtree whose depth is k. This makes it difficult to determine how we simultaneously process each substructure of the input tree. Secondly, MTTs may duplicate trees, which could be a source of inefficiency.

We consider SSUR MTTs so as to avoid the difficulties. The drawback is that their expressiveness is really weak and cannot describe any interesting programs. T^{R}-RELs resolve this drawback. They significantly improve expressiveness because they can tell SSUR MTTs what functions should be applied for each

subtree; indeed, as we will see in Section 5, a T^{R}-REL followed by an SSUR MTT can capture a wide range of tree transformations. Furthermore, efficient parallel evaluation of T^{R}-RELs is fairly straightforward from Theorems 3 and 4.

4.1 Parallel Evaluation of Top-Down Relabeling with Regular Lookaheads

Our parallel evaluation of T^{R}-RELs consists of three steps. First, based on Theorem 3, we associate each node with states of the tree automata for lookaheads; then, based on Theorem 4, we associate each node with the recursive function that take charge of the node. Finally, we rename all constructors in parallel.

Consider $A2B_{\mathrm{REL}}$ as an example.

First, for each node, we calculate whether its child results in s_1 or s_2. We can naturally express the run of the automaton by defining mapping h in Theorem 3 as $h_{\mathsf{A}}(s) = s$, $h_{\mathsf{B}}(s) = s_1$, and $h_{\mathsf{N}} = s_2$. Then, it satisfies the premise of Theorem 3 by taking $\mathcal{F} = \{s_1, s_2\} \to \{s_1, s_2\}$. Note that each element in $\mathcal{F}$ is a constant-time function.

Next, based on Theorem 4, we calculate which function takes charge for each node. In this case, the initial value is rel_0 and mapping h is $h_{(\mathsf{A},s_1)}(rel) = rel_{\mathsf{A}}$, $h_{(\mathsf{A},s_2)}(rel) = rel$, and $h_{(\mathsf{B},s)}(rel) = rel$, where s is either s_1 or s_2. Then, the premise of Theorem 4 holds by taking $\mathcal{F} = \{rel_0, rel_{\mathsf{A}}\} \to \{rel_0, rel_{\mathsf{A}}\}$.

Now that we know that how each node should be processed, we can rename each constructor in parallel. In this case, we rename (A, s_2, rel_0) to A', $(\mathsf{B}, s_2, rel_{\mathsf{A}})$ to B', and so on.

As seen, parallel evaluation of T^{R}-RELs is possible if the premises of Theorems 3 and 4 are satisfied. In fact, they always hold because we consider finite-state transitions and any transition from a constant-size domain to a constant-size range can be performed in constant time.

Theorem 5. *T^{R}-RELs can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$.

Proof. As discussed, the evaluation mainly contains a bottom-up and a top-down steps. We consider the latter; the former is similar.

Let $(Q, \Sigma, \Delta, q_0, R, \mathcal{A})$ be the T^{R}-REL. We apply Theorem 4. The initial value is q_0. Mapping h is defined by $h_{(\sigma,s_1,\ldots,s_k)}(q) = (q_1, \ldots, q_k)$ for $\sigma \in \Sigma^{(k)}$ if $q(\sigma(x_1, \ldots, x_k))\ \langle s_1, \ldots, s_k \rangle = \delta(q_1\ x_1, \ldots, q_k\ x_k)$ is in R. Note that $s_1, \ldots, s_k$ have been associated in the preceding bottom-up step. This computation satisfies the premise by taking $\mathcal{F} = Q \to Q$ because the size of Q is constant. □

Skillicorn [8] used a similar method for parallel evaluation of XML path queries; yet, he only considered the top-down step.

4.2 Parallel Evaluation of Strongly Single-Use Restricted Macro Tree Transducers

It is more difficult to deal with SSUR MTTs. In order to obtain an intuition, we borrow data-flow diagrams from attribute grammars [9]. Figure 2 shows a data-flow diagram for $Flatten_{\mathrm{ssur}}$. The up and the down arrows respectively denote return values and accumulative arguments.

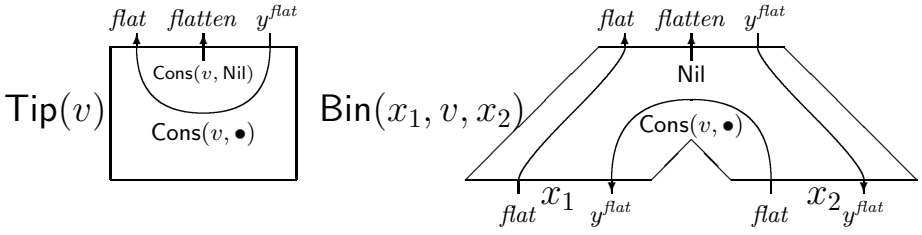


Fig. 2. A data-flow diagram for $\mathit{Flatten}_{\mathrm{ssur}}$ with Tip and Bin. y^f denotes the accumulative argument for function f, and f denotes the return value of function f. $\mathsf{Cons}(v, \bullet)$ means an operation $(\lambda x.\ \mathsf{Cons}(v, x))$.

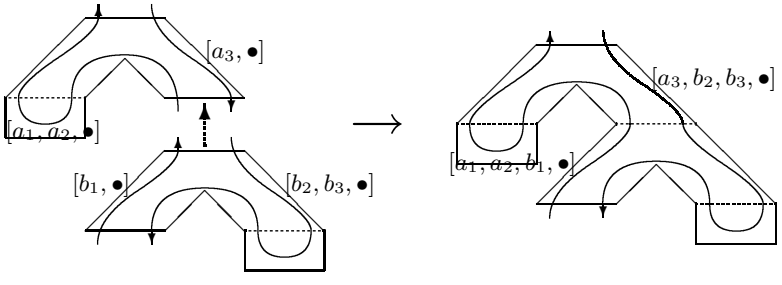


Fig. 3. A Compress operation for $\mathit{Flatten}_{\mathrm{ssur}}$. $[a_1, \ldots, a_n, \bullet]$ means an operation $(\lambda x.\ \mathsf{Cons}(a_1, \mathsf{Cons}(\cdots(a_n, x)\cdots)))$.

A diagram can be seen as a circuit. For example, the diagram for Tip takes one input, y^{flat}, and outputs two values, *flat* and *flatten*. *flatten* results in $\mathsf{Cons}(v, \mathsf{Nil})$. The result of *flat* is calculated by adding v to the value given from y^{flat}.

Well, consider applying Theorem 2. We use mapping h that results in the data-flow diagram for each node. Each primitive contraction operation connects diagrams. A Rake operation fills a missing part of a diagram. A Compress operation, as depicted in Fig. 3, connects two diagrams each of which has exactly two missing parts, one is the parent and the other is a child.

We shrink the diagrams (i.e., circuits) while connecting them. For example, consider the up-arrows in Fig. 3. Originally, each diagram contains a gate that updates the given value of *flat* and passes it to its parent: the lower adds b, and the upper adds a_2 and a_1. After connecting the diagrams, we can see that these consecutive gates are equivalent to a gate that adds b_2, a_2, and a_1. In this way, we can keep diagrams simple. Each diagram consists of O(1) gates each of which represents an O(1) operation.

This approach is applicable for all SSUR MTTs. Since each recursive call occurs at most once, we can easily translate an SSUR MTT to a data-flow diagram. It is not the case in those that are not SSUR. For instance, function *exp* does two recursive calls with different accumulative parameters, making it impossible to describe it as a data-flow diagram. The SSUR property also ensures

that the arrows do not fork. Moreover, each gate in the diagrams represents an application of a context, and thus we can efficiently merge consecutive gates.

Theorem 6. *SSUR MTTs can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$.

Proof. Let $(Q, \Sigma, \Delta, q_0, R)$ be the SSUR MTT, $m+1$ and k be the maximum rank of symbols in Q and Σ, respectively, $Q_k = \{q_i \mid q \in Q \wedge 1 \leq i \leq k\}$, $X_k = \{x_i \mid 1 \leq i \leq k\}$, $Y_m = \{y_i \mid 1 \leq i \leq m\}$, $Y_m^Q = \{y_i^q \mid y_i \in Y_m \wedge q \in Q\}$, and $Y_{m,k}^Q = \{y_{i,j}^q \mid y_i^q \in Y_m^Q \wedge 1 \leq j \leq k\}$. Given term $t \in T_{\Delta \cup Q}(X_k \cup Y_m)$, which is a subterm of a right-hand-side expression in R, let $\lceil t \rceil_q \in T_\Delta(Q_k \cup Y_m^Q)$ be the term obtained by replacing each subterm of t of the form $q'(x_i, e_1, \ldots, e_{m'})$ $(q' \in Q^{(m'+1)})$ by q'_i and y_i by y_i^q.

We define set S_σ $(\sigma \in \Sigma)$, which corresponds to the data-flow diagram for σ, as follows. For each $q(\sigma(x_1, \ldots, x_{k'}), y_1, \ldots, y_{m'}) = e$ in R, S_σ contains $q := \lceil e \rceil_q$; besides, for each occurrence of $q'(x_i, e_1, \ldots, e_{m'})$ in the right-hand side of $q(\sigma(x_1, \ldots, x_{k'}), y_1, \ldots, y_{m''}) = e$ in R, S_σ contains each of $y_{j,i}^{q'} := \lceil e_j \rceil_q$ $(1 \leq j \leq m')$. Thus, S_σ consists of terms of the form $v := t$, where $v \in Q \cup Y_{m,k}^Q$ and $t \in T_\Delta(Q_k \cup Y_m^Q)$. It is worth noting that they exactly correspond to attribute grammars: Q and Y_m^Q are the sets of synthesized and inherited attributes, respectively, and S_σ is the semantic rule for σ.

We apply Theorem 2. Mapping h is defined by $h_\sigma = S_\sigma$. For each primitive contraction operation, we merge the associated sets, i.e., the term representations of the data-flow diagrams, as follows. Assume that a contraction operation is applied to the n-th child. Let S_p and S_c be the sets associated with the parent and the child, respectively. Let $\Rightarrow_p$ and $\Rightarrow_c$ be a rewriting by the sets of rewrite rules, $\{q_n \to e_1[y_i^q \mapsto e_2]_{(y_{i,n}^q := e_2) \in S_p} \mid (q := e_1) \in S_c\}$ and $\{y_i^q \to e_1[p_n \mapsto e_2]_{(p := e_2) \in S_c} \mid (y_{i,n}^q := e_1) \in S_p\}$, respectively. Let $\mathrm{NF}(\Rightarrow_*, e)$ to be the normal form of e under the rewriting by $\Rightarrow_*$, where $*$ is either p or c. We associate the node left after the contraction operation with the following S^*:

$$\begin{aligned} S^* = {} & \{q := \mathrm{NF}(\Rightarrow_p, e) \mid (q := e) \in S_p\} \cup \\ & \{y_{i,j}^q := \mathrm{NF}(\Rightarrow_p, e) \mid (y_{i,j}^q := e) \in S_p \wedge j \neq n\} \cup \\ & \{y_{i,n}^q := \mathrm{NF}(\Rightarrow_c, e) \mid (y_{i,j}^q := e) \in S_c\} \end{aligned}$$

After the tree contraction is finished, we extract an element of the form $q_0 := t$ from the associated set, and return t as the output. This procedure calculates the correct result in the required time bound. Note that the calculation of the normal form can be done in constant time. The SSUR property ensures that during the rewriting, the right-hand-side of $v := t$ contains no v and at most one occurrence of each variable. □

The parallel evaluation described above may calculate trees that are to be discarded. Such cases occur when accumulating parameters or return values of some functions are not used. As seen in the top-down step of the parallel evaluation of T^{R}-RELs, a preprocessing can determine which function should be evaluated for each node. Then, we can estimate variables that will not contribute the final result and thereby reduce construction of useless trees.

5 Parallel Evaluation of Complicated Tree Transformations of Linear Size Increase

We can enjoy parallelism if we write our programs by T^{R}-RELs and SSUR MTTs. One may feel that T^{R}-RELs and SSUR MTTs are too weak; however, and surprisingly, they cover a large class of interesting tree transformations.

In the following, we use the power notation: for class of tree transformation F, $F^1 = F$ and $F^{n+1} = \{f \circ f' \mid f \in F \wedge f' \in F^n\}$.

The key notion is the *linear size increase* [10,11].

Definition 6. *Tree transformation $f \in T_1 \to T_2$ is said to be of* linear size increase *(abbreviated to LSI) if there exists constant c such that $|f(t)| \leq c \cdot |t|$ holds for all $t \in T_1$. The class of LSI tree transformations is denoted by LSI.* □

LSI implies that the transformation will not result in a significantly larger tree than the input. For example, *Flatten* and *A2B* are LSI, while *Exp* is not.

Most tree operations that we would like to evaluate in parallel seem to fall into LSI. A typical case is that the input tree is huge. Then, it is impossible to achieve the transformation if its output is much larger than the input. Therefore, capturing LSI tree transformations would be meaningful.

5.1 Macro Tree Transformations

It is possible to check whether a composition of MTTs is LSI (Theorem 2 of [11]). Moreover, our theorems, in combination with those by Engelfriet and Maneth [6, 10,11], enable us to evaluate them efficiently in parallel.

Theorem 7. *Any tree transformation in $MTT^n \cap LSI$ ($n \in \mathbb{N}$) can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$.

Proof. In the following, *MSOTT* denotes the set of monadic second-order logic definable tree transformations [12].

$$
\begin{aligned}
MTT^n \cap LSI = {} & \{ \text{ Theorem 1 of [11] } \} \\
& MTT \cap LSI \\
= {} & \{ \text{ Theorem 7.2 of [10] } \} \\
& MSOTT \\
= {} & \{ \text{ Theorems 5.10 and 7.1 of [6] } \} \\
& MTT_{\mathrm{ssur}} \circ T^{\mathrm{R}}\text{-REL}
\end{aligned}
$$

Note that, for each equation, we can effectively construct the corresponding instance. Therefore, we are able to obtain an SSUR MTT and a T^{R}-REL, and then Theorems 5 and 6 enable us to perform their parallel evaluation. □

This theorem leads to parallel evaluation of a lot of tree transformations. It is worth stressing that, as mentioned in the proof, we are really able to construct an SSUR MTT and a T^{R}-REL from a program described by MTTs.

5.2 Higher-Order Tree Transformations

MTTs are first-order. It cannot take tree-generating function as a parameter. High level tree transducers [13] extend MTTs in such a way that their accumulative parameter may retain contexts.

Informally, high level tree transducers are defined as follows. Trees are 1-level values. A context that takes and results in k-level values is $(k+1)$-level value. A k-level tree transducer is an obvious extension of MTTs and could have accumulative parameters that retain values whose levels are at most k.

We can evaluate high level tree transducers in parallel if they are LSI.

Theorem 8. *Any high level tree transducer can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$ *if it is LSI.*

Proof. It is immediate from Theorem 7 and the fact that the class of tree transformations realized by k-level tree transducers is MTT^k (Corollary 4.13 of [1] and Theorem 8.2 of [13]). □

5.3 Primitive Recursive Tree Transformations

An MTT can only describe an iteration over the input. Therefore, it cannot express, for instance, $\mathit{flat}_{\mathrm{para}}$ discussed in the introduction, in which a recursive function, $+\!\!+$, repeatedly traverses the output of flat'. Modular tree transducers [14] extend MTTs. They allow this kind of nested recursions and thereby compute exactly the class of primitive recursive functions on trees.

Unfortunately, modular tree transducers are strictly more expressive than finite compositions of MTTs. Yet, Engelfriet and Vogler [14] proved that modular tree transducers can be described by MTTs if they are *calling-restricted*. Informally, a calling-restricted modular tree transducer consists of functions each of which has a module number that is larger than those of functions that generate its first argument. For example, $f(\sigma(x_1, \ldots, x_k)) = \cdots g(h(e)) \cdots$ satisfies the restriction only if the module number of g is larger than that of h.

Theorem 9. *Any calling-restricted modular tree transducer can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$ *if it is LSI.*

Proof. It is immediate from Theorem 7 and the fact that each calling-restricted k-modular tree transducer is in MTT^{k+1} (Corollary 7.8 of [14]). □

5.4 Tree-Walking Transformations with Pebbles

A possible application of our result is XML processing. Since XML data could be very large, parallel processing would be useful. Pebble tree transducers [15] are introduced so as to model XML transformations, and they can express most of the practical XML transformations expressible without joins.

A k-pebble tree transducer generates its output by walking (i.e., going either up or down) in the input with putting and lifting k pebbles on nodes.

It can determine its behavior depending on whether pebbles are on the current node. In spite of their rather complicated definition, Engelfriet and Maneth [16] proved that pebble tree transducers can be expressed by compositions of MTTs. Therefore, we obtain the following theorem.

Theorem 10. *Any pebble tree transducer can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$ *if it is LSI.*

Proof. It is immediate from Theorem 7 and the fact that each k-pebble tree transducer is in MTT^{k+1} (Theorem 35 of [16]). □

5.5 Other Classes of Tree Transformations

In addition to those we have mentioned, there are several classes of tree transformations that we can perform parallel evaluation if they are LSI. For example, we can deal with attributed tree transducers [17], which describe attribute-grammars-like tree transformations, because each attributed tree transducer is in MTT [2]. MSO tree transducers [12] transform trees by using queries written in monadic second-order logic formulae. MSO tree transducers are proved to be equivalent to $MTT \cap LSI$ [6] and thus parallelizable. Our method is also applicable to macro forest tree transducers [18], which provide another model of XML transformations, because each macro forest transducer is in MTT^2 [18].

Moreover, we can deal with those that are described by compositions of tree transformations in these classes. The following corollary summarizes our results. We omit MTTs because some other classes, such as high-level tree transducers and calling-restricted modular tree transducers, include them.

Corollary 1. *Any tree transformation that is described by a composition of high-level tree transducers, calling-restricted modular tree transducers, pebble tree transducers, attributed tree transducers, MSO tree transducers, and macro forest transducers can be evaluated in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$ *if it is LSI.* □

6 Concluding Remarks

We have discussed parallel evaluation of tree transformations and shown that a large class of LSI tree transformations can be efficiently evaluated in parallel. Though our initial motivation was to develop a parallel evaluation method for accumulative functions, known facts enable us to deal with more functions including a large subset of the primitive recursions on trees. Our result is based on the theory of parallel tree contraction and that of MTTs. To the best of the author's knowledge, this is the first study that connects them.

There have been several studies on parallel evaluation of accumulative tree-operating functions [4, 5, 19, 8, 20, 21, 22]. They are based on two computation patterns, called upward accumulations and downward accumulations [5], which respectively correspond to Theorems 3 and 4. They are useful for describing bottom-up and top-down accumulative computations, but not for left-to-right, right-to-left, or more complicated tree traversals.

In order to deal with complicated tree traversals, we have borrowed intuitions from attribute grammars and combined them with parallel tree contraction. This is the technical contribution of this paper. Parallel evaluation of attribute grammars has been discussed [23, 24, 25, 27, 26, 28, 29]. Most studies consider simultaneously evaluating values of independent attributes, whereas we consider merging partially calculated results, namely contexts or more generally semantic rules. A notable exception is scan grammars by Reps [28]. Scan grammars have a special instruction called *scan*. The *scan* instruction specifies that an attribution is calculated by an associative operator and enables us to compute the value efficiently in parallel. Our method is related to scan grammars in the sense that it relies on the associativity of substitution. The crucial difference is that our method can deal with complicated tree traversals, while scan grammars can only describe simple left-to-right or right-to-left traversals.

We believe that usefulness of the combination of attribute grammars and parallel tree contraction is not restricted to tree transformations. It could be a useful method for parallel evaluation of accumulative functional programs that may consist of operations besides constructors. This is a future work.

Another future work is to improve practical efficiency. Our result may not be practical because derivation of an SSUR MTT and a T^R-REL from given programs could result in unreasonably complicated functions and thereby will introduce very large constant factors. A possible solution is deriving a series of SSUR MTTs and T^R-RELs each of which are sufficiently simple. This approach might work well if the given programs are well structured, such as XPath-based XML processing.

Lastly, we briefly discuss efficient implementation. The balanced-tree-based implementation of parallel tree contraction [30] would be best suitable for our purpose. The method translates parallel tree contraction to bottom-up or top-down sweeps on self-balancing trees. Since the self-balancing nature keeps the height of the tree $O(\log N)$, each bottom-up or top-down sweep can be performed in time $O(N/P + \log P)$. The method has several advantages. It avoids destructing or reconstructing the original tree, is easier to implement downward and upward accumulations, and its ability of rebalancing is useful for further parallel operations on calculated trees. Moreover, it indicates that by distributing independent subtrees to processors, our method could be adapted for distributed-memory environments. If we neglect the time required for distributing the input and gathering the output, our method runs in the same asymptotic time complexity. This approach works well especially when we distribute the input in advance, as distributed XML databases, and results of tree transformations are kept to be distributed to each machine until gathering is explicitly required. Yet, we should be careful that even though all processors originally retain nearly the same number of nodes, the sizes of the output fragments may be quite different.

Acknowledgement. The author is grateful to Keisuke Nakano who suggested studying parallel evaluation of MTTs, Shigeyuki Sato and Kiminori Matsuzaki for collaborative discussions, and anonymous reviewers for helpful comments.

References

1. Engelfriet, J., Vogler, H.: Macro Tree Transducers. J. Comput. Syst. Sci. 31(1), 71–146 (1985)
2. Fulöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers. Springer-Verlag New York, Inc. (1998)
3. Reif, J.H. (ed.): Synthesis of Parallel Algorithms. Morgan Kaufmann Publishers (1993)
4. Abrahamson, K.R., Dadoun, N., Kirkpatrick, D.G., Przytycka, T.M.: A Simple Parallel Tree Contraction Algorithm. J. Algorithms 10(2), 287–302 (1989)
5. Gibbons, J., Cai, W., Skillicorn, D.B.: Efficient Parallel Algorithms for Tree Accumulations. Sci. Comput. Program. 23(1), 1–18 (1994)
6. Engelfriet, J., Maneth, S.: Macro Tree Transducers, Attribute Grammars, and MSO Definable Tree Translations. Inf. Comput. 154(1), 34–91 (1999)
7. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007), http://www.grappa.univ-lille3.fr/tata
8. Skillicorn, D.B.: Structured Parallel Computation in Structured Documents. J. UCS 3(1), 42–68 (1997)
9. Knuth, D.E.: Semantics of Context-Free Languages. Math. Syst. Theor. 2(2), 127–145 (1968)
10. Engelfriet, J., Maneth, S.: Macro Tree Translations of Linear Size Increase Are MSO Definable. SIAM J. Comput. 32(4), 950–1006 (2003)
11. Maneth, S.: The Macro Tree Transducer Hierarchy Collapses for Functions of Linear Size Increase. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 326–337. Springer, Heidelberg (2003)
12. Bloem, R., Engelfriet, J.: A Comparison of Tree Transductions Defined by Monadic Second Order Logic and by Attribute Grammars. J. Comput. Syst. Sci. 61(1), 1–50 (2000)
13. Engelfriet, J., Vogler, H.: High Level Tree Transducers and Iterated Pushdown Tree Transducers. Acta Inf. 26(1/2), 131–192 (1988)
14. Engelfriet, J., Vogler, H.: Modular Tree Transducers. Theor. Comput. Sci. 78(2), 267–303 (1991)
15. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML Transformers. J. Comput. Syst. Sci. 66(1), 66–97 (2003)
16. Engelfriet, J., Maneth, S.: A Comparison of Pebble Tree Transducers with Macro Tree Transducers. Acta Inf. 39(9), 613–698 (2003)
17. Fülöp, Z.: On Attributed Tree Transducers. Acta Cybern. 5, 261–279 (1981)
18. Perst, T., Seidl, H.: Macro Forest Transducers. Inf. Process. Lett. 89(3), 141–149 (2004)
19. Skillicorn, D.B.: Parallel Implementation of Tree Skeletons. J. Parallel Distrib. Comput. 39(2), 115–125 (1996)
20. Hu, Z., Takeichi, M., Iwasaki, H.: Diffusion: Calculating Efficient Parallel Programs. In: Proc. the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pp. 85–94. ACM, New York (1999)
21. Matsuzaki, K., Hu, Z., Takeichi, M.: Parallelization with Tree Skeletons. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 789–798. Springer, Heidelberg (2003)
22. Matsuzaki, K., Hu, Z., Takeichi, M.: Parallel Skeletons for Manipulating General Trees. Parallel Comput. 32(7-8), 590–603 (2006)

23. Boehm, H.J., Zwaenepoel, W.: Parallel Attribute Grammar Evaluation. In: Proc. the 7th International Conference on Distributed Computing Systems, pp. 347–355. IEEE Computer Society, Los Alamitos (1987)
24. Kuiper, M.F., Swierstra, S.D.: Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism. In: Deransart, P., Jourdan, M. (eds.) Attribute Grammars and their Applications. LNCS, vol. 461, pp. 61–75. Springer, Heidelberg (1990)
25. Jourdan, M.: A Survey of Parallel Attribute Evaluation Methods. In: Alblas, H., Melichar, B. (eds.) SAGA School 1991. LNCS, vol. 545, pp. 234–255. Springer, Heidelberg (1991)
26. Klaiber, A.C., Gokhale, M.: Parallel Evaluation of Attribute Grammars. IEEE Trans. Parallel Distrib. Syst. 3(2), 206–220 (1992)
27. Klein, E.: Parallel Ordered Attribute Grammars. In: Proc. the 1992 International Conference on Computer Languages, pp. 106–116. IEEE Computer Society, Los Alamitos (1992)
28. Reps, T.W.: Scan Grammars: Parallel Attribute Evaluation via Data-Parallelism. In: Proc. the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 367–376. ACM, New York (1993)
29. Saraiva, J., Henriques, P.: Concurrent Attribute Evaluation. Comput. Syst. Eng. 6(4-5), 451–457 (1995)
30. Morihata, A., Matsuzaki, K.: Balanced Trees Inhabiting Functional Parallel Programming. In: The 16th ACM SIGPLAN International Conference on Functional Programming (accepted, 2011)

Decentralized Delimited Release

Jonas Magazinius[1], Aslan Askarov[2], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology
[2] Cornell University

Abstract. Decentralization is a major challenge for secure computing. In a decentralized setting, principals are free to distrust each other. The key challenge is to provide support for expressing and enforcing expressive decentralized policies. This paper focuses on *declassification* policies, i.e., policies for intended information release. We propose a decentralized language-independent framework for expressing what information can be released. The framework enables combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it. We instantiate the framework for a simple imperative language to show how the decentralized declassification policies can be enforced by a runtime monitor and discuss a prototype that secures programs by inlining the monitor in the code.

1 Introduction

Decentralization is a major challenge for secure computing. In a decentralized setting, principals are free to distrust each other. The key challenge is to provide support for expressing and enforcing expressive decentralized policies. Decentralization is of major concern for language-based information-flow security [42]. Information-flow security ensures that the flow of data through program constructs is secure. Information-flow based techniques are helpful for establishing end-to-end security. For example, a common security goal is *noninterference* [16, 21, 42, 48] that demands that public output does not depend on secret input. There has been much progress on tracking information flow in languages of increasing complexity [42], and, consequently, information-flow security tools for languages such as Java, ML, and Ada have emerged [36, 38, 47].

A particularly important problem in the context of information-flow security is *declassification* [46] policies, i.e., policies for intended information release. These policies are intended to allow some information release as long as the information release mechanisms are not abused to reveal information that is not intended for release. Revealing the result of a password check is an example of intended information release, while revealing the actual password is unintended release. Similarly, the average grade for an exam is an example of intended information release, while revealing the individual grades of all students is unintended release. Abusing the underlying declassification mechanism for unintended release constitutes *information laundering*.

Decentralization makes declassification particularly intriguing. When is a piece of data allowed to be released? The answer might be simple when the piece of data originates from a single principal and needs to be passed to another one. However, when the piece of data originates from several sources, data release needs to satisfy security

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 220–237, 2011.

requirements of all parties involved. Despite a large body of work on declassification (discussed in Section 5), providing a clean semantic treatment for decentralized declassification has been so far out of reach. Concretely, the unresolved challenge we address is prevention of information laundering in decentralized security policies.

Consider a scenario of a web mashup. A *web mashup* is a web service that integrates a number of independent services into a single web service. A common example is a mashup that combines information on available apartments and a map service (such as Google Maps) in an interactive service that displays apartments for sale on a map. Components of a mashup typically originate from different Internet domains.

A crucial challenge when building secure mashups [17] is hitting the sweet spot between separation and integration. The components need to communicate with each other but without stealing sensitive information. For example, a mashup that displays trucks with dangerous goods on a map might reveal the corner points of a required map to the map service but it must not reveal sensitive information about displayed objects such as the type of dangerous goods [26].

Collaboration in the presence of mutual distrust requires solid policy and enforcement support. Pushing the mashup scenario further, consider two web services (say, Gmail and Facebook) that are willing to swap sensitive information under the condition that both provide their share. For example, this might be a client-side mashup that allows cross-importing Gmail's and Facebook's address books. We want the policy framework to support the swap but prevent stealing Gmail's address book by Facebook.

A prominent line of work on declassification in a decentralized setting is the *decentralized label model (DLM)* [32]. This model underlies the security labels tracked by the Java-based information-flow tracker Jif [36]. DLM labels explicitly records owners. Owners are allowed to introduce arbitrary declassification on the part of labels they own. However, no soundness arguments for Jif's treatment of the labels are provided.

While inspired by DLM, our goal is precise semantic specification of decentralized security and its sound enforcement. Our focus is on exactly what can be released, which prevents information laundering. Unlike the DLM enforcement as performed by Jif, we do distinguish between programs that reveal the result of matching against a password from programs that reveal the password itself.

Combining the decentralization in the fashion of DLM and the laundering prevention in the fashion of *delimited release* [43], this paper proposes a decentralized language-independent framework for expressing what information can be released. The framework enables release of combination of data owned by different principals without compromising their respective security policies. A key feature is that information release is permitted only when the owners of the data agree on releasing it.

To illustrate that the framework is realizable at language level, we instantiate the framework for a simple imperative language to show how the decentralized declassification policies can be enforced by a runtime monitor. We resolve the challenge of respecting decentralized policies while at the same time preventing laundering. Further, the monitor allows on-the-fly addition of new declassification polices by different principles. The monitor provides a safe approximation for the security policy. As it is often the case with automatic enforcement of nontrivial policies, the monitor is incomplete in the sense that some secure runs are blocked.

Further, we have implemented a prototype for a small subset of JavaScript that secures programs by inlining the information-flow monitor in the code. The inlining transformation transforms an arbitrary, possibly insecure, program into one that performs inlined information-flow checks, so that the result of the transformation is secure by construction.

2 Decentralized Delimited Release

Principals and Security Levels. Our model is built upon a notion of *decentralized principals* which we denote via p, q. We assume that principals are mutually distrusting and that there are no "actsfor" or "speaks-for" relations [23, 33] between them.

We consider a *lattice of security levels* $\mathcal{L}$ and denote by $\sqsubseteq$ the ordering between elements of the lattice. A simple security lattice consists of two elements L and H, such that $L \sqsubseteq H$ i.e., L is no more restrictive than H. The structure of the security lattice does not have to be connected to principals in general, though they may be related as illustrated in Section 2.2.

We assume that different parts of global state (or memory) are labeled with different security levels: the higher the security level, the more sensitive the information which is labeled with that level. We also associate every security level in our model with an adversary that may observe memory states at that level: the higher the security level, the more powerful the adversary associated with that level. For two-level security lattice, an adversary corresponding to level L can observe only *low* (or public) parts of the state, while adversary corresponding to level H can observe all parts of the state.

Policies as Equivalence Relations. Our model uses *partial equivalence relations* (PERs) over memories for use in confidentiality policies [1, 45]. The PER representation allows for fine granularity in individual policies. We believe that intentional models of security such as DLM [33] or tag-based models [12, 19, 22, 49] can be easily interpreted using PERs. Section 2.2 is an example of one such translation for a simple subset of DLM.

Intuitively, two memories m and m' are indistinguishable according to an equivalence relation I if $m \ I \ m'$. Two particular relations that we use are Id and All introduced by the following definition:

Definition 1 (Id and All relations). *Assuming that $\mathcal{M}$ ranges over all possible memories, define $Id \triangleq \{(m, m) \mid m \in \mathcal{M}\}$ and $All \triangleq \{(m, m') \mid m, m' \in \mathcal{M}\}$*

Assume an extension of memory mappings from variables to expressions, so that $m(e)$ corresponds to the value of expression e in memory m. We also introduce an indistinguishability induced by a particular set of expressions.

Definition 2 (Indistinguishability induced by $\mathcal{E}$). *Given a set of expressions $\mathcal{E}$, define* indistinguishability induced by $\mathcal{E}$ *as* $\mathsf{Ind}(\mathcal{E}) \triangleq \{(m, m') \mid \forall e \in \mathcal{E} \,.\, m(e) = m'(e)\}$.

In set-theoretical terminology, operator $\mathsf{Ind}(\mathcal{E})$ is the *kernel* of the function that maps memories to values according to a given expression. When $\mathcal{E}$ consists of a single expression e we often write $\mathsf{Ind}(e)$ instead of $\mathsf{Ind}(\{e\})$.

Restriction. We define an operator of *restriction* induced by a set of variables. The operator is handy in the following examples and in the translation in Section 2.1.

Definition 3 (Restriction induced by variables X). *Given a set of variables X, define* restriction induced by X *to be a relation* $\mathsf{S}(X) \triangleq \mathsf{Ind}(\{y \mid y \notin X\})$ *i.e., indistinguishability relation for all variables y that are different from the ones in X.*

It can be easily shown that for disjoint sets of variables X and Y it holds that $\mathsf{S}(X \cup Y) = \mathsf{S}(X) \cup \mathsf{S}(Y)$. We often omit the set notation and write $\mathsf{S}(x, y)$ for $\mathsf{S}(\{x, y\})$.

Example: Consider memory with three variables x, y and z, and relation $\mathsf{S}(z)$. According to Def. 3, $\mathsf{S}(z) = \mathsf{Ind}(\{x, y\}) = \{m, m' \mid m(x) = m'(x) \wedge m(y) = m'(y)\}$. Here $\mathsf{S}(z)$ relates memories that must agree on all variables but z. In particular, given memories m_1 in which $x \mapsto 1, y \mapsto 1, z \mapsto 1$, m_2 in which $x \mapsto 1, y \mapsto 1, z \mapsto 0$, and m_3 in which $x \mapsto 1, y \mapsto 2, z \mapsto 1$ we have that $m_1\ \mathsf{S}(z)\ m_2$ but not $m_1\ \mathsf{S}(z)\ m_3$.

Confidentiality Policies. *Confidentiality policy* is a mapping from security levels in $\mathcal{L}$ to corresponding indistinguishability relations. Consider an example security lattice consisting of three security levels L, M, H, such that $L \sqsubseteq M \sqsubseteq H$. Assume also that our memory contains two variables x and y, and consider a confidentiality policy I such that $I(L) = All, I(M) = \mathsf{S}(x)$, and $I(H) = Id$

According to this policy, an attacker at level L can observe no part of the state, which is specified by $I(L) = All$. An attacker at level M can not observe the value of x but may observe the value of y. This is specified by using a restriction induced by x for $I(M)$. Finally, $I(H)$ establishes that an attacker at level H can observe all variables.

Say that a confidentiality policy I is *well-formed* when $I(\top) = Id$, where $\top$ is the most restrictive element in $\mathcal{L}$. Moreover, for any two labels $\ell \sqsubseteq \ell'$ it must hold that $I(\ell) \supseteq I(\ell')$. Our example policy above is well-formed. Indeed, $I(H) = Id$ and $I(L) = All \supseteq I(M) = \mathsf{S}(x) = \mathsf{Ind}(y) \supseteq I(H) = Id$. It is also easy to show that a policy obtained from point-wise union and intersection of well-formed policies is well-formed. The rest of the paper assumes that all policies are well-formed.

2.1 Decentralized Policies

In a decentralized setting every principal provides its confidentiality policy. We denote a confidentiality policy of principal p as I_p. In particular, $I_p(\ell)$ is a relation specifying what memories must be indistinguishable at levels ℓ and below according to principal p. Given two principals p and q with policies I_p and I_q, the combination of these policies is policy I' s.t. for all ℓ we have $I'(\ell) = I_p(\ell) \cup I_q(\ell)$. Note that I' combines restrictions of both p and q and is as restrictive as both I_p and I_q. The following definition generalizes combination of trusted policies.

Definition 4 (Combination of confidentiality policies). *Given a number of principals $p_1 \ldots p_n$ with policies I_{p_i}, $1 \leq i \leq n$, the combination of these policies is a policy I' such that for all ℓ it holds that $I'(\ell) = \bigcup_i I_{p_i}(\ell)$.*

Example: Consider a lattice with three levels L, M, and H as before and a memory with two variables x and y. Consider two principals p and q with the policies $I_p(L) = All, I_p(M) = \mathsf{Ind}(x)$, $I_p(H) = Id$, $I_q(L) = All, I_q(M) = \mathsf{Ind}(y), I_q(H) = Id$.

According to these policies p and q have different views on what can be observable at level M. Combining these two policies, we obtain a policy I', such that $I'(L) = All, I'(M) = All$, and $I'(H) = Id$. Combining restrictions of both p and q means $I'(M)$ allows an attacker at level M to observe neither x nor y.

Declassification. Declassification corresponds to relaxing individual policies I_p. We assume that every principal provides a set of *escape hatches* [43] that correspond to that principal's view on what data can be declassified.

Definition 5 (Escape hatches). *An escape hatch is a pair (e, ℓ) where e is a declassification expression, and ℓ is a level to which e may be declassified.*

Given a set of escape hatches $\mathcal{E}_p$ for principal p and an initial indistinguishability policy of this principal I_p we can obtain a less restrictive indistinguishability policy as follows.

Definition 6. *Given a confidentiality policy I and a set of escape hatches $\mathcal{E}$, let* declassification operator D *return a policy that relaxes I with $\mathcal{E}$. We define D pointwise for every level ℓ so that $\mathsf{D}(I, \mathcal{E})(\ell) \triangleq I(\ell) \cap \mathsf{Ind}(\mathcal{E}_\ell)$ where $\mathcal{E}_\ell = \{e \mid (e, \ell') \in \mathcal{E} \wedge \ell' \sqsubseteq \ell\}$ is the selection of escape hatches from $\mathcal{E}$ that are observable at ℓ.*

Example: Consider I_p as in Section 2.1 and escape hatch (y, L). Let us assume $I' = \mathsf{D}(I_p, \{(y, L)\})$. We have $I'(L) = \mathsf{Ind}(x), I'(M) = Id$, and $I'(H) = Id$.

Example: declassification and composite policies. Consider again memory with two variables x and y, a simple two-level security lattice with security levels L and H such that $L \sqsubseteq H$, and two principals p and q. Assume that p's policy specifies that a low attacker cannot observe x, and that q specifies that low observer cannot observe any parts of the memory. The corresponding security policies can be given by the second and third columns of Figure 1, where $\mathsf{S}(x) = \mathsf{Ind}(y)$. The combination of policies of both p and q at level L is $\mathsf{Ind}(y) \cup All = All$. That is, principals agree on no information being observable to an adversary at the level L.

ℓ	$I_p(\ell)$	$I_q(\ell)$	$\mathsf{D}(I_p, \mathcal{E}_p)(\ell)$	$\mathsf{D}(I_q, \mathcal{E}_q)(\ell)$
H	Id	Id	Id	Id
L	$\mathsf{S}(x)$	All	$\mathsf{S}(x) \cap \mathsf{Ind}(x)$	$\mathsf{Ind}(x + y)$

Fig. 1. Declassification and composite policies

Assume principal p declassifies the value of x to L, and principal q declassifies the value of $x + y$ to L, i.e., $\mathcal{E}_p = \{(x, L)\}$ and $\mathcal{E}_q = \{(x + y, L)\}$. The corresponding policies are given by the last two columns of Figure 1. The result of combining policies at level L is captured by the relation $(\mathsf{Ind}(y) \cap \mathsf{Ind}(x)) \cup \mathsf{Ind}(x + y)$ which is equivalent to $\mathsf{Ind}(x + y)$. That is, both principals allow $x + y$ to be observed at level L.

Security. Our security condition is based on decentralized confidentiality policies. For generality, this section uses an abstract notion of a *system with memory*, denoted by $S(\cdot)$. A transition of system $S(m)$ with memory m to a *final* state with memory m' is written as $S(m) \Downarrow m'$. Section 3 instantiates this abstraction with standard program configurations. We call our security condition *decentralized delimited release* (DDR).

Definition 7 (Batch-style DDR). *Assume principals $p_1, \ldots, p_n$ with confidentiality policies $I_1 \ldots I_n$ and declassification policies given by escape hatch sets $\mathcal{E}_1 \ldots \mathcal{E}_n$. Say*

that a system with memory $S(\cdot)$ satisfies decentralized delimited release *when for every level ℓ and for all memories m_1, m_2 for which $m_1 \bigcup_{1\leq i\leq n} \mathsf{D}(I_i, \mathcal{E}_i)(\ell)\ m_2$ it holds that whenever $S(m_1) \Downarrow m_1'$ and $S(m_2) \Downarrow m_2'$ it must be that $m_1' \bigcup_i I_i(\ell)\ m_2'$.*

DDR borrows its intuition from the original definition of delimited release [43], and generalizes it to the case of several principles. In fact, in case of a single principal this definition matches the original definition in [43].

The key element of this definition is that it prevents *laundering* attacks. To see an example of a laundering attack, consider the following examples. Assume a memory with three variables x, y, z and individual policies of two principals p and q, as shown in the second and third columns of Figure 2. Here $\mathsf{S}(x, y)$ is restriction induced by x and y, and $\mathsf{S}(x, y) = \mathsf{Ind}(z)$, i.e., this relation allows only variable z to be observable.

Assume escape hatch sets where p declassifies $x + y$ to L, i.e., $\mathcal{E}_p = \{(x + y, L)\}$, and q declassifies both x and y individually to L, i.e., $\mathcal{E}_q = \{(x, L), (y, L)\}$. Taking the escape hatches into account we obtain the relations shown by the last two columns of Figure 2. According to these policies the program $z := x + y$ is secure. On the other hand the program $x := y; z := x + y$ is insecure. To see this consider two memories m_1 and m_2 where in m_1 we have $x \mapsto 1, y \mapsto 1, z \mapsto 0$ and in m_2 we have $x \mapsto 0, y \mapsto 2, z \mapsto 0$. We have that $m_1\ \mathsf{D}(I_p, \mathcal{E}_p)(L) \cup \mathsf{D}(I_q, \mathcal{E}_q)(L)\ m_2$, but not $m_1'\ I_p(L) \cup I_q(L)\ m_2'$.

DLM^0 . We adopt the Decentralized Label Model (DLM) [32] as our model of expressing security policies sans actsfor relation, that we dub DLM^0 . We nevertheless, retain top and bottom principals $\bot$ and $\top$ that allow us to express the most and the least restrictive security policies. In DLM a security level of a variable records *policy owners*, reviewed below. On the intuitive level policy owner is a principal who cares about the sensitivity of the data. This is more than simply a principal who can read data — not every principal who reads data is necessarily interested in preserving its confidentiality.

DLM policies are the basic building blocks for expressing security restrictions by principals. A (confidentiality) policy is written $o \rightarrow r_1, \ldots, r_n$, where o is the owner of the policy, and $r_1, \ldots, r_n$ is the set of readers. Here principal o restricts the flow of data to the principals in the readers set. For example, in the policy $Alice \rightarrow Bob, Carol$ Alice constraints the set of readers to only Bob, Carol, and herself (the owner is implicitly a reader). Similarly, a policy $Carol \rightarrow Carol$ restricts all but Carol from reading data.

Security labels, denoted by ℓ, are either DLM policies or are composed from other labels in one of the two ways: (i) conjunction of two labels, written $\ell_1 \sqcup \ell_2$, is a label that enforces restrictions of both ℓ_1 and ℓ_2. (ii) disjunction of two labels, written $\ell_1 \sqcap \ell_2$, is a label that enforces restrictions of either ℓ_1 or ℓ_2. An example of a conjunction label is $\{Alice \rightarrow Bob, Carol\} \sqcup \{Carol \rightarrow Carol\}$. Carol is the only reader; because of

ℓ	$I_p(\ell)$	$I_q(\ell)$	$\mathsf{D}(I_p, \mathcal{E}_p)(\ell)$	$\mathsf{D}(I_q, \mathcal{E}_q)(\ell)$
H	Id	Id	Id	Id
L	$\mathsf{S}(x, y)$	$\mathsf{S}(x, y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x + y)$	$\mathsf{S}(x, y) \cap \mathsf{Ind}(x) \cap \mathsf{Ind}(y)$

Fig. 2. Policies for example laundering attack

the Carol's policy, this label restricts either Alice or Bob from reading data. Disjunction label $\{Alice \rightarrow Alice\} \sqcap \{Bob \rightarrow Bob\}$ allows both Alice and Bob to read data.

Labels can be ordered by the "no more restrictive than" [14, 34] relation: $\ell_1 \sqsubseteq \ell_2$ when ℓ_1 restricts data no more than ℓ_2 does. We use $\{\bot \rightarrow \bot\}$ to denote the least restrictive label (also denoted simply $\bot$), i.e., for all ℓ it holds that $\{\bot \rightarrow \bot\} \sqsubseteq \ell$. For example, $\{Alice \rightarrow Alice, Bob\} \sqsubseteq \{Alice \rightarrow Alice\}$, because in the right label, Alice imposes stricter restrictions by allowing only her to be the reader. However (assuming there is no actsfor relationship between Alice and Bob), $\{Alice \rightarrow Bob\} \not\sqsubseteq \{Bob \rightarrow Alice\}$. Here Alice's constraints are not satisfied. Her label on the left restricts the flow to Bob, but there are no Alice's policies on the right.

2.2 From DLM^0 to Families of Indistinguishability Relations

This section shows how DLM^0 labels can be translated to confidentiality policies. The translation is parametrized by the principals. We define two operators in this translation — the top level translation operator $\widetilde{\mathsf{T}}_p$ and a helper operator T_p. The top level translation operator $\widetilde{\mathsf{T}}_p$, that returns a confidentiality policy for principal p, takes the variable environment Γ as a single argument. It is defined so that when $\Gamma = \emptyset$ then in the resulting confidentiality policy $\widetilde{\mathsf{T}}_p(\Gamma)$, the corresponding indistinguishability relation for all labels ℓ is Id. This indeed matches the DLM intuition that no restrictions imply the most permissive confidentiality policy. To translate restrictions that are captured by DLM labels, we define a helper operator $\mathsf{T}_p(I_p, \ell, x)$.

Definition 8 (Translation of a single label T_p). *Given a principal p with an initial policy I_p, label ℓ, and variable x, define $\mathsf{T}_p(I_p, \ell, x)$ inductively based on the structure of ℓ.*

case ℓ **is an empty label** *Return* I_p.
case ℓ **is** $\ell' \sqcup \{q \rightarrow \boldsymbol{r}\}$ **such that** $q \neq p$ **and** $q \neq \top$ *Return* $\mathsf{T}_p(I_p, \ell', x)$.
case ℓ **is** $\ell' \sqcup \{q \rightarrow \boldsymbol{r}\}$ **such that** $q = p$ **or** $q = \top$ *Define policy I'_p, where for all ℓ'' let*

$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup \mathsf{S}(x) & \text{if } \{q \rightarrow \boldsymbol{r}\} \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases} \quad \text{and return } \mathsf{T}_p(I'_p, \ell', x).$$

case ℓ **is** $\ell' \sqcap \{q \rightarrow \boldsymbol{r}\}$ **such that** $q = p$ **or** $q = \top$ *Define policy I'_p where for all ℓ'' let*

$$I'_p(\ell'') = \begin{cases} I_p(\ell'') \cup \mathsf{S}(x) & \text{if } \{q \rightarrow \boldsymbol{r}\} \not\sqsubseteq \ell'' \wedge \ell' \not\sqsubseteq \ell'' \\ I_p(\ell'') & \text{otherwise} \end{cases} \quad \text{and return } \mathsf{T}_p(I'_p, \ell', x).$$

With the definition of T_p at hand we define the top-level translation operator $\widetilde{\mathsf{T}}_p$.

Definition 9 (Translation of DLM^0 policies). *Assume that Γ maps variables to DLM^0 labels. Define an operator $\widetilde{\mathsf{T}}_p$ that translates restrictions recorded in Γ to confidentiality policies as follows. We let $\widetilde{\mathsf{T}}_p(\emptyset) = \widetilde{Id}$, when $\Gamma = \emptyset$, and otherwise $\widetilde{\mathsf{T}}_p(x \mapsto \ell; \Gamma') = \mathsf{T}_p(\widetilde{\mathsf{T}}_p(\Gamma'), \ell, x)$. Here $\widetilde{Id}$ is a policy s.t. for all levels ℓ it holds $\widetilde{Id}(\ell) = Id$.*

Example: Consider memory consisting of four variables x, y, z and w. Assume two principals p and q, and variable environment Γ, s.t. $\Gamma(x) = \{p \rightarrow p\}, \Gamma(y) = \{q \rightarrow q\}, \Gamma(z) = \{p \rightarrow p, q\} \sqcup \{q \rightarrow p, q\}$, and $\Gamma(w) = \{p \rightarrow p\} \sqcap \{q \rightarrow q\}$. Translation of labels in Γ is represented by the second and third columns in the table below.

ℓ	$\widetilde{T}_p(\Gamma)(\ell)$	$\widetilde{T}_q(\Gamma)(\ell)$	$\mathsf{D}(\widetilde{T}_p(\Gamma), \mathcal{E}_p)(\ell)$	$\mathsf{D}(\widetilde{T}_q(\Gamma), \mathcal{E}_q)(\ell)$
$\{\top \rightarrow \top\}$	Id	Id	Id	Id
$\{p \rightarrow p\}$	Id	$\mathsf{S}(y)$	Id	$\mathsf{S}(y) \cap \mathsf{Ind}(x+y)$
$\{q \rightarrow q\}$	$\mathsf{S}(x)$	Id	$\mathsf{S}(x) \cap \mathsf{Ind}(x+y)$	Id
$\{p \rightarrow p, q\} \sqcup \{q \rightarrow p, q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$	$\mathsf{S}(x) \cap \mathsf{Ind}(x+y)$	$\mathsf{S}(y) \cap \mathsf{Ind}(x+y)$
$\{p \rightarrow p\} \sqcap \{q \rightarrow q\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$	$\mathsf{S}(x)$	$\mathsf{S}(y)$
$\{\bot \rightarrow \bot\}$	$\mathsf{S}(x)$	$\mathsf{S}(y)$	$\mathsf{S}(x)$	$\mathsf{S}(y)$

Here $\mathsf{S}(x) = \mathsf{Ind}(y) \cap \mathsf{Ind}(z) \cap \mathsf{Ind}(w)$ and $\mathsf{S}(y) = \mathsf{Ind}(x) \cap \mathsf{Ind}(z) \cap \mathsf{Ind}(w)$. Consider escape hatches provided by each principals such that $\mathcal{E}_p = \mathcal{E}_q = \{(x+y, \{p \rightarrow p, q\} \sqcup \{q \rightarrow p, q\})\}$. Taking escape hatches into account the policies obtained from declassification operator are illustrated in fourth and fifth columns of the table above.

3 Enforcement

This section illustrates the realizability of our framework for a simple imperative language. We formalize the language along with a runtime enforcement mechanism that ensures security.

Language and Semantics. The syntax of the language is displayed in Figure 3. Expressions e operate on values n and variables x and might involve composition with operator op. Commands c are standard imperative commands. The only nonstandard primitive in the language is a declassification primitive $\texttt{declassify}(p, e, \ell)$ that declares an escape hatch (e, ℓ) of principal p.

Figure 4 contains the semantic rules for evaluating commands. A *memory* is a mapping from variables to values, where values range over some fixed set of values (say, without loss of generality, the set of integers). We assume an extension of memories to expressions that is computed using a semantic interpretation of constants as values and operators as total functions on values. This allows us writing $m(e)$ for the value of expression e in memory m. A *configuration* has the form $\langle c, m\rangle$ where c is a command in the language and m is a memory. A *transition* has the form $\langle c, m\rangle \xrightarrow{\beta} \langle c', m'\rangle$ representing a computation step from configuration $\langle c, m\rangle$ to $\langle c', m'\rangle$. *Events* β are there to communicate relevant information about program execution to an execution monitor (this style of presenting monitors follows recent work on information-flow monitoring, e.g., [5, 44]). When events are unimportant, we may omit explicitly writing them out as in $\langle c, m\rangle \longrightarrow \langle c', m'\rangle$. The meaning of the particular events is spelled out in the description of the monitor below.

Monitor. Our enforcement mechanism is a runtime monitor. Listening to a given program event, the monitor either grants execution (possibly updating its internal state) or blocks it. Following the idea sketched in [26], we obtain security by requiring two conditions on declassification (in addition to standard tracking "regular" flows orthogonal to declassification). The first condition is to check that all declassifications are allowed. The second condition ensures that the value of an escape hatch expression has not changed since the start of the program. The former is in charge of the *who* dimension of declassification, preventing release to unauthorized principals, whereas

$$e ::= n \mid x \mid e \; op \; e$$
$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{declassify}(p, e, \ell) \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } e \texttt{ do } c$$

Fig. 3. Syntax

$$\langle \texttt{declassify}(p, e, \ell), m\rangle \xrightarrow{d(p,e,\ell)} \langle stop, m\rangle \qquad \frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{a(x,e,m)} \langle stop, m[x \mapsto v]\rangle}$$

$$\frac{m(e) = n \qquad n \neq 0 \implies i = 1 \qquad n = 0 \implies i = 2}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{b(e)} \langle c_i; end, m\rangle} \qquad \langle end, m\rangle \xrightarrow{f} \langle stop, m\rangle$$

Fig. 4. Monitored semantics: selected rules

$$\langle st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{b(e)} \langle lev(e) : st, i, \mathcal{E}, \Gamma\rangle \qquad \langle hd : st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{f} \langle st, i, \mathcal{E}, \Gamma\rangle$$

$$\langle st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{d(p,e,\ell)} \langle st, i, \mathcal{E}[p \mapsto \mathcal{E}_p \cup \{(e, \ell, lev(st))\}], \Gamma\rangle$$

$$\frac{lev(st) \sqsubseteq \Gamma(x) \qquad \ell \triangleq \mathsf{substEH}(lev(e), x, e, \mathcal{E}, \Gamma) \qquad lev(e) \not\sqsubseteq \ell \implies m(e) = i(e)}{\langle st, i, \mathcal{E}, \Gamma\rangle \xrightarrow{a(x,e,m)} \langle st, i, \mathcal{E}, \Gamma[x \mapsto lev(st) \sqcup \ell]\rangle}$$

$$\begin{aligned}
\mathsf{substEH}(\{\mathsf{o} \to \widetilde{\mathsf{r}}\}, \mathsf{x}, \mathsf{e}, \mathcal{E}, \Gamma) &\triangleq \{o \to \widetilde{r}\} \sqcap \{\ell \mid (e, \ell, pc) \in \mathcal{E}_o \wedge pc \sqsubseteq \Gamma(x)\} \\
\mathsf{substEH}(\ell_1 \sqcup \ell_2, x, e, \mathcal{E}, \Gamma) &\triangleq \mathsf{substEH}(\ell_1, x, e, \mathcal{E}, \Gamma) \sqcup \mathsf{substEH}(\ell_2, x, e, \mathcal{E}, \Gamma) \\
\mathsf{substEH}(\ell_1 \sqcap \ell_2, x, e, \mathcal{E}, \Gamma) &\triangleq \mathsf{substEH}(\ell_1, x, e, \mathcal{E}, \Gamma) \sqcap \mathsf{substEH}(\ell_2, x, e, \mathcal{E}, \Gamma)
\end{aligned}$$

Fig. 5. Monitor semantics: selected rules

the latter controls the *what* dimension, preventing information laundering. Section 5 discuses these and other dimensions of declassification [46] in further detail.

Figure 5 presents selected monitor rules. Monitor configurations have the form $\langle st, i, \mathcal{E}, \Gamma\rangle$, where st is a stack of security levels, i stores the initial program memory, $\mathcal{E}$ is an indexed collection of sets of escape hatches, and Γ is the current security environment. Escape hatches are also extended to the form (e, ℓ, pc), where pc records the level of the monitor stack when that escape hatch has been added. The monitor features a form of flow-sensitivity: security level of a variable $\Gamma(x)$ can be updated, but only when the decision to update does not give away secret information [6].

Assume an overloaded function $lev(\cdot)$ that returns the least upper bound on the security level of components in the argument. For expressions, the components are the subexpressions and for lists the components are the list elements. When monitor stack is empty $lev(\cdot)$ is the least restrictive label $\bot \to \bot$.

The event $b(e)$ is generated by conditionals and loops when branching on an expression e. This is interesting information for the monitor because it introduces risks for *implicit information flow* [18] through control-flow structure of the program. For example, program $\texttt{if } h \texttt{ then } l := 1 \texttt{ else } l := 0$ leaks whether the initial value of (secret) variable h is (non)zero into the final value of (public) variable l. The essence of an implicit flow is a public side effect in a secret computation context. To record the computation context, we keep track of the security levels of the variables branched on. Thus, the monitor always accepts branching on an expression, pushing the level of the expression on the stack. The event f is generated by conditionals and loops on reach-

ing a joint point of branching. The monitor always accepts this event, popping the top security level from the stack. The event $d(p, e, \ell)$ is generated upon declassification of expression e to level ℓ by principal p. In response, the monitor includes the newly declared escape hatch in its environment and records the current level of the stack $lev(st)$.

The event $a(x, e, m)$ is generated by assignment of an expression e to a variable x in memory m. First, the monitor blocks implicit flows by requiring that the level of the x is at least as restrictive as the least upper bound of the security levels on the stack. Next, the monitor checks if this assignment can be treated as a declassification. The operator substEH performs a label substitution and returns the least restrictive label that can be obtained by using declassifications in $\mathcal{E}$. Note that all information used by substEH check is bounded by $\Gamma(x)$ — we only look up escape hatches that syntactically agree on expression e and that were updated in the contexts with $pc \sqsubseteq \Gamma(x)$. If expression can be declassified to a level that is more permissive than $lev(e)$, the monitor checks that the escape-hatch expression must be the same in the initial and current memories. This prevents information laundering as in $\texttt{declassify}(p, h, p \rightarrow \bot); h := h'; l := h$ where h is declared to be declassified whereas h' is actually leaked. Finally, the monitor updates the level of $\Gamma(x)$, featuring flow-sensitivity mentioned earlier in this Section.

The monitor accepts program $l := x + y$, if both A's and B's escape hatches contain $x + y$, and rejects it if either A or B do not explicitly list $x + y$ in their escape hatches.

While, as we will show, the enforcement is sound, it is obviously incomplete. In the setting of the example above, the program is rejected when A's escape-hatch set is $\{x\}$ and B's is $\{y\}$. A and B are willing to release all of their data, and so the program is rightfully accepted secure by the security definition. However, the monitor rejects the program because expression $x + y$ is not found in the escape-hatch sets.

Soundness. The monitor guarantees secure execution in the presence of mutual distrust. We instantiate the notion of system with memories of Definition 7 with monitored program configurations $(\langle c, m\rangle, \langle st, i, \mathcal{E}, \Gamma\rangle)$. Assume all declassification policies are expressed in $\mathcal{E}$ and c contains no further `declassify` statements. This is consistent with our implementation (cf. Section 4) in which escape hatches are collected at parse time. We write $(\langle c, m\rangle, \langle st, i, \mathcal{E}, \Gamma\rangle) \Downarrow m', \Gamma'$ when $(\langle c, m\rangle, \langle st, i, \mathcal{E}, \Gamma\rangle) \longrightarrow^{*} (\langle stop, m'\rangle, \langle st', i, \mathcal{E}, \Gamma'\rangle)$, where $\longrightarrow^{*}$ is a transitive closure of $\longrightarrow$. Assume principals $p_1, \ldots, p_n$ with individual declassification policies $\mathcal{E}_{p_i}$. Formally, we have:

Theorem 1 (Soundness). *Assume principals $p_1, \ldots, p_n$ with initial* DLM^0 *policies expressed in the environment Γ and declassification policies expressed by the collection of sets of escape hatches $\mathcal{E}$, indexed by p_i. Consider program c free of* `declassify` *statements. Then for all levels ℓ and memories m_1, m_2 s.t. $m_1 \bigcup_p \mathsf{D}(\widetilde{\mathsf{T}}_p(\Gamma), \mathcal{E}_p)(\ell) m_2$ if $(\langle c, m_1\rangle, \langle \epsilon, m_1, \mathcal{E}, \Gamma\rangle) \Downarrow m_1', \Gamma_1'$, and $(\langle c, m_2\rangle, \langle \epsilon, m_2, \mathcal{E}, \Gamma\rangle) \Downarrow m_2', \Gamma_2'$, then $\bigcup_p \widetilde{\mathsf{T}}_p(\Gamma_1')(\ell) = \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma_2')(\ell)$ and $m_1' \bigcup_p \widetilde{\mathsf{T}}_p(\Gamma_1')(\ell) m_2'$.*

The proof of Theorem 1 is available in the accompanying technical report [27].

Example: We revisit the example with aggregate computation from Section 2.1. Consider variable environment consisting of three variables x, y and z. Assume two

principals p and q s.t. $\Gamma(x) = \{p \to p\}, \Gamma(y) = \{q \to q\}$, and $\Gamma(z) = \{p \to p, q\} \sqcup \{q \to p, q\}$. and escape hatch sets for every principal s.t. $\mathcal{E}_p = \mathcal{E}_q = \{(x + y, \{p \to p, q\} \sqcup \{q \to p, q\}, \bot \to \bot)\}$. Then basic declassification of the form $z := x + y$ is accepted, while laundering as in the program $x := y; z := x + y$ is rejected.

4 Experiments

Next, we present the experiments conducted on enforcement of the monitor in practice. The inlining transformation converts a program in a language from Section 3 into a program in JavaScript with inlined security checks. In this experiment we have successfully implemented two scenarios in a restricted subset of JavaScript.

Experiment Setup. To implement runtime source transformation we need functionality for parsing and rewriting of JavaScript code, written in JavaScript. We use ANTLR [2] to generate such a parser/rewriter from a JavaScript grammar. The generated parser is 7650 LOC of JavaScript, not counting additional 165 LOC for the user-defined JavaScript and 6139 LOC in the runtime library. For performance, the code can be further reduced using JavaScript compression tools. All sources are available on demand.

The monitor must be inlined before the code is parsed by the browser, or else the code is executed unmonitored. The Opera browser [37] allows the user to include privileged JavaScript called "User JavaScript". User JavaScript can access functions and events not accessible to ordinary JavaScript, including the event "BeforeScript", that enables rewriting the script source before the source reaches the browser's parser. This allows us to inline the monitor whenever a new script is loaded.

This approach introduces two sources of runtime overhead. The first is the parsing and rewriting, performed once per code segment. The second is the execution of the inlined monitor. Previous work [29] shows the total overhead of 2–10 times the untransformed runtime, depending on the code structure of, the browser, and the system used.

One alternative to implementing the monitor is using aspect-oriented techniques along the lines of, e.g., [28]. However, such an implementation would demand low-level access to program operations. For example, performing an assignment or reaching a joint point must be observable events in order to serve as pointcuts.

Transformation. The generated parser parses and, in the process, rewrites the code, transforming it on the fly. If the parser cannot parse the input it throws an error and the code is not evaluated by the browser. The monitored code is hence limited by the parser.

The source language is a subset of JavaScript, as described in Section 3. The target language is full JavaScript. This means there are no restrictions on the monitor itself, only on the code being monitored. We identify different stages in the transformation that are closely related to the stages of the browser as it requests content. While other JavaScript-specific features, such as prototyping and objects, would make an interesting complement, more research on how such features affect information-flow analysis is required before extending the language and incorporating them in the framework.

Transformation in Stages. Based on information available at a given moment, only certain actions can be taken. Thus, we distinguish between *parse-time* and *run-time*.

Parse-Time. As scripts are encountered we enumerate their origins and for each origin load the associated escape hatches and initial levels for variables. The scripts are parsed on the fly. During parsing, when a security critical part of the source is encountered, we rewrite the source inlining the monitor according to a set of rules. Because JavaScript lacks a declassification primitive, unlike the monitor in Section 3, escape hatches are defined at parse-time. Note that while it is clear at parse-time which variables are used in an expression, their run-time values are unknown. This is crucial for declassification as it relies on which variables are used in expressions to determine which information to declassify. This transformation is detailed below.

Run-Time. At run-time, as the program is evaluated, all variables have their actual values, but when following an execution path we lose information about the control-flow structure of the program. Thus, the inlining transformation needs to encode necessary control-flow structure information for the monitor. As the transformed script is executed, the monitor validates the inlined checks.

```
var x;    // User variable
var _x_; // Level of x
var __x; // Initial value of x
var _pc; // Special variable
```

Listing 1.1. Naming convention

Shadow Variables. To track information flow in the program we use shadow variables. Two kinds of shadow variables are used: one for the level of the variable, and one for its initial value. The shadow variables that hold the initial values are set when the corresponding variable is declared, while the shadow variable that hold the level are updated whenever the corresponding variables are initially assigned. The set of shadow variables corresponds to Γ in the formal monitor. Also, a small set of monitor specific variables is described below.

To prevent the code being monitored from interfering with state of the monitor, the shadow variables must be isolated. One could create a separate namespace for shadow variables, with minimal impact on the source program. The drawback is mimicking the scoping and variable lookup mechanisms of JavaScript, to prevent clashes between equally named variables from different scopes. Implementing this can be non-trivial.

Antoher possibility is to reserve an infrequently used character, such as "_", for shadow variables, thereby excluding it from the set of allowed characters for identifiers in the source language. This would prevent valid code, according to the parser, from referring to variables using this character. The benefit in this case is that we can piggy-back on JavaScripts built in scoping mechanism. The drawback is that we moderately restrict the set of valid programs. As a design choice, we chose this option. The chosen naming convention can be seen below in Figure 1.1.

Special Variables. A few special variables exist to store the state of the monitor at run-time. For tracking implicit informations flows, the level of the cusrrent execution context is stored in the special variable `_pc`. The `_pc` works like a stack and is updated whenever a new execution context is entered. The variable `_E` stores all escape hatches and their associated levels. Finally the variable `_init` stores all initial levels of variables as defined by the owner of each variable.

Transformation Rules. We focus on the interesting cases of the transformation: assignment, declassification, and branching.

Assignment and Declassification. Following the semantics in Figure 4, the transformed code updates both the value of variable being assigned and the level of the corresponding shadow variable. Which level it updates to depend on whether the assignment expression is in the set of escape hatch expressions or not. In the case of declassification, the level is determined from the escape hatch, otherwise the new level is determined from the variables used in the expression. When determining the level, the current level of the execution context (the `_pc`) is also considered.

```
// Implicit flow check
while(!_pc.leq(_x_));
if ('y+z' in _E) {
  // Laundering check
    while((__y+__z)!=(y+z));
    _x_=_pc.join(_E['y+z']);
} else {
    _x_=_pc.join(_y_,_z_)
}
x=y+z;
```

Listing 1.2. Assignment rule

Insecure upgrade refers to assignment of a lower level variable in a higher level context, implying an information leak [40]. Insecure upgrade is prevented by checking that the `_pc` is less than or equal to the level of the variable [6]. If it is not, the program gets stuck. Information laundering through declassification is prevented by checking that the current value is the same as the initial value of the expression. If this check fails, the program gets stuck. Listing 1.2 gives an example of an assignment after transformation.

Branches. To prevent implicit information flows, the monitor tracks the level of the context in each branch. When a branch is encountered, the current level of the `_pc` is stored. The `_pc` is updated with the join of its current level and the level of the expression that is branched upon. Each of the two alternative code paths are transformed and after the two branches join again, the level of the `_pc` before the branch is restored. In the implementation, management of the `_pc` is done through helper methods, e.g. `_pc.branch(_x_); if(x){...}; _pc.joinPoint();`.

Scenarios. We have applied the transformation to two simple yet illustrative scenarios. We believe that the approach of using inline transformation and escape hatches for tracking information flow scales to more complex scenarios: no matter how complex the language is, the secure use of escape hatches is restricted to simple patterns (with no modification of data involved in them).

```
while(!_pc.leq(_d_));
if ('orderOf(f)/p' in _E) {
   while((orderOf(__f)/__p)!=
      (orderOf(f)/p));
   _d_=_pc.join(_E['orderOf(f)/p']);
} else
   _d_=_pc.join(_f_,_p_);
d=orderOf(f)/(10*p);
```

Listing 1.3. Scenario 1 transformed

Social E-Commercing. In this scenario we have an e-commerce site (A) and a social networking site (B) who have an agreement that the users of the social networking site get a discount (d) on the products of the e-commerce site if they recommend the store to their friends. The size of the discount is determined by the price (p) and the number of friends (f) that the user recommends the site to. To protect the privacy of the user, the social networking site does not want to release the exact number of friends so the discount is calculated by the following

```
while(!_pc.leq(_a_));
if ('a.concat(b)' in _E) {
   while((__a.concat(__b))!=
      (a.concat(b)));
   _a_=_pc.join(_E['a.concat(b)']);
} else
   _a_=_pc.join(_a_,_b_);
a = a.concat(b);
```

Listing 1.4. Scenario 2 transformed

formula: $d = e(f,p) = \frac{orderOf(f)}{10*p}$. For declassification the A specifies the escape hatch $E(A) = \{(e(f,p), \bot)\}$. An example of the transformed code for this scenario is available in Listing 1.3. In this scenario A could maliciously try to find the exact number of friend recommendations, e.g. using either `var x=f;` or `while(x<f)x++;`. Regardless, since both explicit and implicit information-flows are tracked this information is labeled as belonging to B.

Contact Swap. Consider a mashup where the user can synchronize his contact lists on several social networking sites. In this scenario we have a truly distributed and collaborative release of information. The sites need to collaborate on which contacts to share and whom to share them with. That is, the user might be unwilling to share the contacts marked as business associates across networks, but still want to share contacts marked as friends. A sample of the transformed scenario code is available in Listing 1.4. Here both A and B would need to declassify the expression `a.concat(b)` to the other. As can be seen in this sample, the rewritten code prevents potential attacks. Malicious code could try to launder some other information by assigning it to either `a` or `b`, as such `b= secret; a=a.concat(b);`. However, the transformation of this code gets stuck in the initial value check since the value of b no longer matches its initial value.

5 Related Work

There is a large body of work on declassification, much of which is discussed in Sabelfeld and Sands' recent overview [46]. The overview presents dimensions and principles of declassification. The identified dimensions correspond to *what* data is released, *where* and *when* in the program and by *whom*. The *what* and *where* dimensions and their combinations have been studied particularly intensively [4, 5, 8, 9, 30].

Our approach integrates the *what* and *who* dimensions. It is the *who* dimension that has received relatively little attention so far. The precursor to work on the *who* dimension in the language-based setting is the decentralized label model (DLM) [32]. DLM allows principals expressing ownership information as well as explicit read/write access lists in security labels. Chen and Chong [11] generalizes DLM to describe a range of owned policies from information flow and access control to software licensing.

Work on *robustness* [3, 35], addressed the *who* dimension by preventing attacker-controlled data from affecting *what* is released. Lux and Mantel [24] investigate a bisimulation-based condition that helps expressing who (or, more precisely, what input channels) may affect declassification.

Our approach builds on the *composite release* [26] policy that combines the *what* and *who* dimensions. The escape hatches express the *what* and the ownership of the principals of the escape-hatch policies expresses the *who*. However, for composite release to be allowed, the principals have to *syntactically* agree on escape hatches. This paper removes this limitation and generalizes the principal model to handle DLM. The experimental part is another added value with respect to the previous work [26].

Broberg and Sands [10] describe *paralocks*, a knowledge-based framework for expressing declassification and role-based access-control policies. Broberg and Sands show how to encode DLM's *actsFor* relation using paralocks. However, paralocks do not address the *what* dimension of declassification.

Our enforcement draws on the ideas sketched by us earlier [26], where we present considerations for practical enforcement of composite release. The formalization of the enforcement fits well into the modular framework [5, 44] for dynamic information-flow monitoring where the underlying program and monitor communicate through the interface of events. The *what* part of declassification is enforced similarly to [5], by ensuring that the values of escape-hatch expressions have not been modified. The paper extends the formalization of the enforcement with the *who* part.

Recent efforts approach inlining for information flow. Chudnov and Naumann [15] inline a flow-sensitive hybrid monitor by Russo and Sabelfeld [40]. The monitor does not offer support for declassification. As in this work, Magazinius et al. [29] concentrate on inlining purely dynamic monitors under the no-sensitive-upgrade discipline. The distinct feature is inlining on the fly, which allows a smooth treatment of dynamic code evaluation. While the inlining rules [29] offer no support for declassification, it is still a useful starting point for our experiments in Section 4.

In the web setting, work on language-based sadboxing such as object capabilities (e.g., [25, 31]) is less related because separation does not allow information flow and intended release. The most closely related project is the Mozilla project FlowSafe [20] that aims at extending Firefox with runtime information-flow tracking, where dynamic information-flow monitoring [6, 7] lies at its core.

6 Conclusion

We have presented a framework for specitying and enforcing decentralized information-flow policies. The policies express possibilities of collaboration in the environment of mutual distrust. By default, no information flow is allowed across different principals. Whenever principals are willing to collaborate, the policy framework ensures that a piece of data is revealed only if all owners of the data have provided sufficient authorization for the release. While the policy framework is independent, we have demonstrated that is realizable with language support. We have showed how to enforce security by runtime monitoring for a simple imperative language.

A major direction of future work is integrating support for decentralized security policies into the line of work on information-flow controls in a web setting, where we have already investigated the treatment of dynamic code evaluation [5], timeout events [39], and interaction with the DOM tree [41].

Another intriguing avenue for integration is with Chong's *required release* [13] policy. This policy ensures that if a principal promises to release a piece of data, then this piece of data must be released. Such a policy is an excellent fit for thwarting attempts of cheating. For example, suppose three principals have agreed on releasing the average of their three pieces of data to each other. However, a cheating principal might attempt to withdraw its escape hatch or declassify to a level that is not sufficient for the other principals to be able to access the result. These attempts can be prevented by required release, where principals must release data according to the declared policies.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.: A core calculus of dependency. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 147–160 (January 1999)
2. ANTLR Parser Generator, http://www.antlr.org/
3. Askarov, A., Myers, A.: A Semantic Framework for Declassification and Endorsement. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 64–84. Springer, Heidelberg (2010)
4. Askarov, A., Sabelfeld, A.: Localized delimited release: Combining the what and where dimensions of information release. In: Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS), pp. 53–60 (June 2007)
5. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proc. IEEE Computer Security Foundations Symposium (July 2009)
6. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS) (June 2009)
7. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, University of California, Santa Cruz (2009)
8. Banerjee, A., Naumann, D., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Proc. IEEE Symp. on Security and Privacy, pp. 339–353 (May 2008)
9. Barthe, G., Cavadini, S., Rezk, T.: Tractable enforcement of declassification policies. In: Proc. IEEE Computer Security Foundations Symposium (June 2008)
10. Broberg, N., Sands, D.: Paralocks: role-based information flow control and beyond. In: Proc. ACM Symp. on Principles of Programming Languages (January 2010)
11. Chen, H., Chong, S.: Owned policies for information security. In: Proc. IEEE Computer Security Foundations Workshop (June 2004)
12. Cheng, W.: Information Flow for Secure Distributed Applications. PhD thesis, Massachusetts Institute of Technology (September 2009)
13. Chong, S.: Required information release. In: Proc. IEEE Computer Security Foundations Symposium (July 2010)
14. Chong, S., Myers, A.C.: Decentralized robustness. In: Proc. IEEE Computer Security Foundations Workshop, pp. 242–253 (July 2006)
15. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: Proc. IEEE Computer Security Foundations Symposium (July 2010)
16. Cohen, E.S.: Information transmission in sequential programs. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.) Foundations of Secure Computation, pp. 297–335. Academic Press (1978)
17. Decat, M., De Ryck, P., Desmet, L., Piessens, F., Joosen, W.: Towards building secure web mashups. In: Proc. AppSec Research (June 2010)
18. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Comm. of the ACM 20(7), 504–513 (1977)
19. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and event processes in the Asbestos operating system. In: Proc. 20th ACM Symp. on Operating System Principles (SOSP) (October 2005)
20. Eich, B.: Flowsafe: Information flow security for the browser (October 2009), https://wiki.mozilla.org/FlowSafe
21. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Proc. IEEE Symp. on Security and Privacy, pp. 11–20 (April 1982)
22. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Proc. 21st ACM Symp. on Operating System Principles, SOSP (2007)

23. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. In: Proc. ACM Symp. on Operating System Principles, pp. 165–182 (October 1991); Operating System Review 253(5)
24. Lux, A., Mantel, H.: Who Can Declassify? In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 35–49. Springer, Heidelberg (2009)
25. Maffeis, S., Mitchell, J.C., Taly, A.: Object capabilities and isolation of untrusted web applications. In: Proc. IEEE Symp. on Security and Privacy (May 2010)
26. Magazinius, J., Askarov, A., Sabelfeld, A.: A lattice-based approach to mashup security. In: Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS) (April 2010)
27. Magazinius, J., Askarov, A., Sabelfeld, A.: Decentralized delimited release. Technical report, Chalmers University of Technology (2011), http://www.cse.chalmers.se/~d02pulse/ddr-tr.pdf
28. Magazinius, J., Phung, P., Sands, D.: Safe wrappers and sane policies for self protecting javascript. In: Nordic Conference on Secure IT Systems. Springer, Heidelberg (2010)
29. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-Fly Inlining of Dynamic Security Monitors. In: Rannenberg, K., Varadharajan, V., Weber, C. (eds.) SEC 2010. IFIP AICT, vol. 330, pp. 173–186. Springer, Heidelberg (2010)
30. Mantel, H., Reinhard, A.: Controlling the What and Where of Declassification in Language-Based Security. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 141–156. Springer, Heidelberg (2007)
31. Miller, M., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized javascript (2008)
32. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proc. ACM Symp. on Operating System Principles, pp. 129–142 (October 1997)
33. Myers, A.C., Liskov, B.: Complete, safe information flow with decentralized labels. In: Proc. IEEE Symp. on Security and Privacy, pp. 186–197 (May 1998)
34. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology 9(4), 410–442 (2000)
35. Myers, A.C., Sabelfeld, A., Zdancewic, S.: Enforcing robust declassification and qualified robustness. J. Computer Security 14(2), 157–196 (2006)
36. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. Software release. Located (July 2001-2009), http://www.cs.cornell.edu/jif
37. Opera, User JavaScript, http://www.opera.com/docs/userjs/
38. Praxis High Integrity Systems. Sparkada examiner. Software release, http://www.praxis-his.com/sparkada/
39. Russo, A., Sabelfeld, A.: Securing timeout instructions in web applications. In: Proc. IEEE Computer Security Foundations Symposium (July 2009)
40. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Proc. IEEE Computer Security Foundations Symposium (July 2010)
41. Russo, A., Sabelfeld, A., Chudnov, A.: Tracking Information Flow in Dynamic Tree Structures. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 86–103. Springer, Heidelberg (2009)
42. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (2003)
43. Sabelfeld, A., Myers, A.C.: A Model for Delimited Information Release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
44. Sabelfeld, A., Russo, A.: From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 352–365. Springer, Heidelberg (2010)

45. Sabelfeld, A., Sands, D.: A Per Model of Secure Information Flow in Sequential Programs. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 40–58. Springer, Heidelberg (1999)
46. Sabelfeld, A., Sands, D.: Declassification: Dimensions and principles. J. Computer Security 17(5), 517–548 (2009)
47. Simonet, V.: The Flow Caml system. Software release. Located (July 2003), http://cristal.inria.fr/~simonet/soft/flowcaml/
48. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Computer Security 4(3), 167–187 (1996)
49. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: Proc. 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI), pp. 263–278 (2006)

Cost Analysis of Concurrent OO Programs

Elvira Albert[1], Puri Arenas[1], Samir Genaim[1], Miguel Gómez-Zamalloa[1], and German Puebla[2]

[1] DSIC, Complutense University of Madrid, Spain
[2] DLSIIS, Technical University of Madrid, Spain

Abstract. Cost analysis aims at automatically approximating the *resource consumption* (e.g., memory) of executing a program in terms of its input parameters. While cost analysis for sequential programming languages has received considerable attention, concurrency and distribution have been notably less studied. The main challenges (and our contributions) of cost analysis in a concurrent setting are: (1) Inferring precise *size relations* for data in the program in the presence of shared memory. This information is essential for bounding the number of iterations of loops. (2) Distribution suggests that analysis must keep the cost of the diverse distributed components separate. We handle this by means of a novel form of *recurrence equations* which are parametric on the notion of *cost center*, which represents a corresponding component. To the best of our knowledge, our work is the first one to present a general cost analysis framework and an implementation for concurrent OO programs.

1 Introduction

Distribution and concurrency are now mainstream. The Internet and the broad availability of multi-processors radically influence software. Many standard desktop programs have to deal with distribution aspects like network transmission delay and failure. Furthermore, many chip manufactures are turning to multicore processor designs as a way to increase performance in desktop, enterprise, and mobile processors. This brings renewed interest in developing both new concurrency models and associated programming languages techniques that help in understanding, analyzing, and verifying the behavior of concurrent programs.

One of the most important features of a program is its resource consumption. By resource, we mean not only traditional cost measures (e.g., memory consumption) but also concurrency-related measures (e.g., tasks spawned, requests to remote servers). The goal of this paper is to develop a *cost analysis* [22] (a.k.a. *resource usage* analysis) for concurrent OO programs. Cost analysis aims at *statically* inferring approximations of the resource consumption of executing the program. Automatically inferring the resource usage of concurrent programs is challenging because of the inherent complexity of concurrent behaviors.

In addition to traditional applications (e.g., optimization [22], verification and certification of resource consumption [7]), cost analysis opens up interesting applications in the context of concurrent programming. In general, having anticipated knowledge on the resource consumption of the different components

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 238–254, 2011.

which constitute a system is useful for distributing the load of work. Upper bounds (UBs) can be used to predict that one component may receive a large amount of remote requests, while other siblings are idle most of the time. Also, our framework allows instantiating the different components with the particular features of the infrastructure on which they are deployed. Then, analysis can be used to detect the components that consume more resources and may introduce bottlenecks. Lower bounds (LBs) on the resource usage can be used to decide if it is worth executing locally a task or requesting remote execution.

In order to develop our analysis, we consider a concurrency model based on the notion of concurrently running (groups of) objects, similar to the actor-based and active-objects approaches [18, 20]. These models take advantage of the concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. It is recognized that performing the analysis on a high-level concurrency model makes verification more feasible. This is because analysis in concurrent systems often needs to consider too many interleavings and thus ends up being limited to very small programs in practice.

We propose a static cost analysis for concurrent OO programs, which is parametric w.r.t. the notion of resource that can be instantiated to measure both traditional and concurrency-related resources. The main contributions of this work are: (1) We introduce a sound size analysis for concurrent execution. The analysis is *field-sensitive*, i.e., it tracks data stored in the heap whenever it is sound to do so; (2) We lift the definition of cost used in sequential programming to the distributed setting by relying on the notion of *cost centers* [17], which represent the (distributed) components and allow separating their costs; (3) We present a novel form of cost recurrence relations, which is parametric w.r.t. cost centers, but still can be solved to closed-form UBs/LBs using standard solvers for cost analysis of sequential programs; (4) We increase the accuracy of the field-sensitive size analysis by means of *class invariants* [16] which contain information on the shared memory; and (5) We report on a prototype implementation.

2 A Language with Concurrent Objects

The concurrency model of Java and C# is based on threads which share memory and are scheduled preemptively, i.e., they can be suspended or activated at any time. To avoid undesired interleavings, low-level synchronization mechanisms such as locks are used. Thread-based programs are error-prone, difficult to debug, verify and maintain. In order to overcome these problems, several concurrency models that take advantage of the inherent concurrency implicit in the notion of object have been developed [8, 14, 16, 18, 20]. They provide simple language extensions which allow programming concurrent applications with relatively little effort. In this work we focus on the ABS language [13] which inherits the concurrency model of Creol [8, 14] and extends it with the possibility of grouping objects together, as in JCoBoxes [18]. For simplicity, we do not consider object groups and assume that objects are independent, and we exclude

```
data List⟨A⟩=Nil | Cons(A,List⟨A⟩);
data Set⟨A⟩=EmptyS | Insert(A,Set⟨A⟩);
data Pairs⟨A,B⟩=Pair(A,B);
data Map⟨A,B⟩=EmptyM |
        Assoc(Pairs⟨A,B⟩,Map⟨A,B⟩);
type FN, Packet=String;
type FNs=Set⟨String⟩;
type File=List⟨Packet⟩;
type Catalog=List⟨Pairs⟨Node,FNs⟩⟩;
def B lookup⟨A,B⟩(Map⟨A,B⟩ ms, A k)=
  case ms { Assoc(Pair(k,y),_) ⇒ y;
            Assoc(_,tm) ⇒ lookup(tm,k);}
```

```
def Bool contains⟨A⟩(Set⟨A⟩ s,A e)=
  case s {
    EmptyS ⇒ False;
    Insert(e, _) ⇒ True;
    Insert(_, xs) ⇒ contains(xs, e);}
def Node findServer(FN f, Catalog c)=
  case c {
    Nil ⇒ null;
    Cons(Pair(s, fs), r) ⇒
      case contains(fs, f) {
        True ⇒ s;
        False ⇒ findServer(f, r); };}
```

Fig. 1. Functional Sequential Part of ABS Implementation of P2P Network

interfaces and inheritance. However, we handle them in our implementation. An ABS *program* consists of a functional, sequential part (data-type and function) and an imperative, concurrent part (interfaces, classes, and a `main` method). This distinction allows combining encapsulation and data transfer between objects such that: objects are passed by reference and used for asynchronous calls, and functional data is used to transfer information between objects. In order to illustrate our approach, we use a peer-to-peer (P2P) distributed application [14].

Fig. 1 shows (part of) the functional fragment of the P2P program which includes type definitions (*String* and *Int* are predefined) and functions which can be executed in a standard way. *FN* defines a file name as a *String*, and the content of a *File* is defined as a list of *Packet*s. Fig. 2 shows the imperative part of the program. Class Node reflects that peers can act as clients and servers. A P2P network is formed by a set of interconnected peers which make the files stored in their database (an object of class DB) available to other peers, without central coordination. The only coordination is by means of an object of class NetWork, whose code is not shown due to space limitations. It is enough to know that nodes learn who their neighbors are by invoking getNeighbors. A node acting as client triggers computations with searchFile, which first finds a neighbor node s that can provide the file and then requests the file using reqFile, which in turn makes a number of activations of the remote method getPacks on s. Whenever possible, size packets are transferred at a time. The field size is set by the constructor of class Node and remains constant. Fig. 2 shows also a main method that creates a configuration with three nodes, two databases and one administrator.

The central concept of our concurrency model is that of *concurrent object*. Conceptually, each object has a dedicated processor and encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the *this* object, and any other object can only access such fields through method calls. Concurrent objects live in a distributed environment with asynchronous and unordered communication by means of asynchronous method calls. Thus, an object has a set of tasks (i.e., calls) to execute and, among them, at most one task is *active* and the others are *suspended* on a task queue.

```
class DB {
  Map⟨FN,File⟩ dbf;
  DB(Map⟨FN,File⟩ db){dbf = db;}
  File getFile(FN fn) {
    return lookup(dbf,fn); }
  Int lengthDB(FN fn) {
    return length(lookup(dbf,fn)); }
  Unit storeFile(FN fn, File f) {
    dbf=Assoc(Pair(fn,f),dbf); }}
class Node {
  DB db; Catalog catalog;
  List⟨Node⟩ myNeighbors;
  Network admin; Int size;
  Node(DB dbf, Int s) {db=dbf;size=s;}
  Catalog availFiles(List⟨Node⟩ sList) ...
  List⟨Packet⟩ getPacks(FN fn,Int ps,Int n) {
    File f=Nil; File res=Nil;Fut⟨File⟩ ff;
    ff=db ! getFile(fn);await ff?; f=ff.get;
    while(ps>0) {
      res = Cons(nth(f,n+ps-1),res);
      ps=ps-1;}
    return res; }
  Int lengthNode(FN fn) {
    Fut⟨Int⟩ length;
    length=db ! lengthDB(fn);
    await length?; return length.get; }
  Unit reqFile(Node s, FN fn) {
    Fut⟨Int⟩ l1; Fut⟨List⟨Packet⟩⟩ l2;
    File f = Nil; List⟨Packet⟩ ps = Nil;
    Int i = 0; Int incr = 0;
    l1 = s ! lengthNode(fn);
    await l1?; i = l1.get;
    while (i > 0) {
        if (size > i) incr = i; else incr = size;
        i = i - incr;
        l2 = s ! getPacks(fn,incr,i);
        await l2? ; ps = l2.get ;
        f = app(ps,f); }
    db ! storeFile(fn,f); }
  Unit searchFile(FN f) ...}
class NetWork {
  Node node1,node2,node3;
  NetWork(Node n1,Node n2,Node n3)...
  List⟨Node⟩ getNeighbors(Node caller)...}
```

```
1 : db = Assoc(Pair("a.txt", Cons("x", Nil)), Assoc(Pair("b.txt", Cons("y", Nil)), EmptyM))
2 : db1 = new DB(EmptyM) ; db2 = new DB(db);
3 : n1 = new Node(db1, 2) ; n2 = new Node(db1, 1) ; n3 = new Node(db2, 1);
4 : admin = new NetWork(n1, n2, n3);
5 : n1 ! setAdmin(admin) ; n2 ! setAdmin(admin) ; n3 ! setAdmin(admin);
6 : n1 ! searchFile("a.txt") ; n2 ! searchFile("b.txt");
```

Fig. 2. Concurrent Part of ABS Implementation of P2P Network, and the main method

Process scheduling is by default non-deterministic, but controlled by *processor release points* and *future variables* in a cooperative way. After asynchronously calling $f := o\,!\,m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here f is a future variable which refers to a return value which has yet to be computed. There are two operations on future variables, which control external synchronization. First, **`await`** f? suspends the active task (allowing other tasks in the object to be scheduled) unless the future variable f has been assigned a value. Second, the value stored in f can be retrieved using f.**`get`**, which blocks all execution in the object until f gets a value (in case it has not been assigned a value yet). An unconditional **`release`** instruction (not used in the example) suspends the current task and lets a pending task in.

2.1 A Rule-Based Intermediate Language

We develop our analysis on an intermediate representation (IR) similar to those for Java bytecode and .NET [3, 9, 19, 21]. The IR of a method (or function) is

obtained by translating each basic block in its control flow graph (CFG) into a procedure, defined by means of *rules* that adhere to the following grammar:

$r ::= m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n.$
$b ::= x{:=}e \mid this.f{:=}e \mid x{:=}\mathsf{new}\ C \mid \mathtt{call}(ct, m(rec, \bar{x}, \bar{y})) \mid \mathtt{await}\ g \mid \mathtt{release} \mid x{:=}y.\mathsf{get}$
$g ::= true \mid g \wedge g \mid x? \mid e\ op\ e \mid match(x,t) \mid nonmatch(x,t)$
$e ::= \mathsf{null} \mid a \mid t \mid this.f \qquad a ::= x \mid n \mid a{-}a \mid a{+}a \mid a{*}a \mid a/a \qquad t ::= x \mid Co(\bar{t})$

where $op{\in}\{>, =, \geq\}$, $m(this, \bar{x}, \bar{y})$ is the *head* of the rule, *this* is the identifier of the object on which the method or function is executing, g specifies the conditions for the rule to be applicable and $b_1, \ldots, b_n$ is the rule's *body*. Calls are of the form $\mathtt{call}(ct, m(rec, \bar{x}, \bar{y}))$ where $ct \in \{\mathtt{m}, \mathtt{b}, \mathtt{f}\}$ in order to distinguish between calls to methods, intermediate blocks and functions; *rec* is a variable that refers to the receiver object; the variables $\bar{x}$ (resp. $\bar{y}$) are the formal parameters (resp. return values). For blocks and functions, *rec* is always *this*. For methods, $\bar{y}$ is either empty or contains a single output variable. Future variables can be used in `await` instructions but not in rule guards. Guards $match(x,t)$ and $nonmatch(x,t)$ simulate **`case`**-expressions. We assume $x \notin vars(t)$. Note that $match(x,t)$ modifies $vars(t)$ when it succeeds. Terms are constructed using $Co(\bar{t})$, where Co is a data symbol and $\bar{t}$ are the arguments, e.g., $Cons(x,y)$. An instruction $\mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}})$ in ABS is represented in the IR by $\mathsf{new}\ C$ followed by a call to the class constructor with the corresponding parameters $\bar{\mathsf{t}}$. The translation from ABS to IR is (almost) identical to that in [3]. A program in the IR consists of classes, functions, and a main method from which the execution starts. A class C consists of a set of methods and a set of fields $\bar{f}_C$. A method $C.m$ is defined by a set of rules with a single rule named $C.m$ (the method entry). The other rules are intermediate procedures that are used only within the method, with $ct = \mathtt{b}$. A function is a (global) set of rules that is accessible from any method (with $ct = \mathtt{f}$) and, therefore, it cannot access nor modify fields. The main method does not belong to any class. We illustrate the IR by means of the following example.

Example 1. Fig. 3 depicts the IR (left) and the CFG (right) of method reqFile. Loops are extracted in separate CFGs to enable compositional cost analysis. The method is represented by four procedures, *reqFile*, *loop*, *if* and if^c, each of them defined by means of guarded rules. $\overline{inp}$ stands for $\langle this, s, fn, f, ps, i, incr, l_2\rangle$ and $\overline{out}$ for $\langle f, ps, l_2, i, incr\rangle$. Guards in rules state the conditions under which the corresponding blocks in the CFG can be executed. When there is more than one successor in the CFG, we create a *continuation procedure* and a corresponding call in the rule. Blocks in the continuation will in turn be defined by means of (mutually exclusive) guarded rules. As a result of the translation, all forms of iteration in the program are represented by means of *recursive* calls. The parameters of the procedure reqFile are as in the corresponding method, plus the reference to the *this* object. When calling a block, we pass as arguments all local variables that are needed in the block. The heap remains implicit.

$reqFile(\langle this, s, fn\rangle, \langle\rangle) \leftarrow$ ① $f := Nil$,
② $ps := Nil$, ③ $i := 0$,
④ $incr := 0$,
⑤ call(m, $lengthNode(\langle s, fn\rangle, \langle l_1\rangle)$),
⑥ await $l_1?$, ⑦ $i := l_1$.get,
⑧ call(b, $loop(\overline{inp}, \overline{out})$),
$thisDB = this.db$;
⑨ call(m, $storeFile(\langle thisDB, fn, f\rangle, \langle\rangle)$).
$loop(\overline{inp}, \overline{out}) \leftarrow i \leq 0$.
$loop(\overline{inp}, \overline{out}) \leftarrow i > 0$, call(b, $if(\overline{inp}, \overline{out})$).
$if(\overline{inp}, \overline{out}) \leftarrow this.size > i, incr := i,$
call(b, $if^c(\overline{inp}, \overline{out})$).
$if(\overline{inp}, \overline{out}) \leftarrow this.size \leq i, incr := this.size,$
call(b, $if^c(\overline{inp}, \overline{out})$).
$if^c(\overline{inp}, \overline{out}) \leftarrow$ ⑩ $i := i - incr$,
⑪ call(m, $getPacks(\langle s, fn, incr, i\rangle, \langle l_2\rangle)$),
⑫ await $l_2?$, ⑬ $ps := l_2$.get,
⑭ call(f, $app(\langle this, ps, f\rangle, \langle f\rangle)$),
call(b, $loop(\overline{inp}, \overline{out})$).

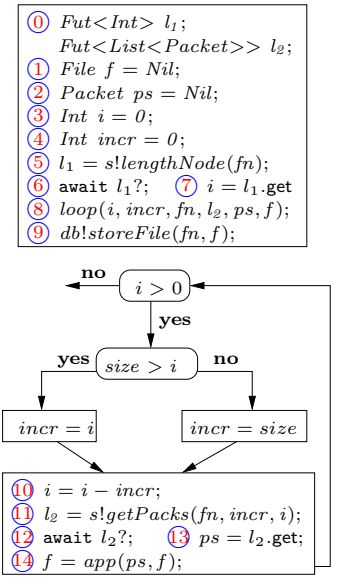


Fig. 3. The IR and CFG for method reqFile

3 Cost and Cost Models for Concurrent Programs

We now define the notion of cost we aim at approximating by static analysis. For this, we need an operational semantics for our language. A state is denoted by $\mathcal{S}$ and includes information on the current state of all objects, including their heaps, active tasks, etc. Each object in $\mathcal{S}$ has a unique identifier. An execution step takes the form $\mathcal{S} \rightsquigarrow_o^b \mathcal{S}'$, in which we move from a state $\mathcal{S}$ to a state $\mathcal{S}'$ by executing instruction b in object o. Thus, we assume a concurrent interleaving semantics. Traces take the form $t \equiv \mathcal{S}_0 \rightsquigarrow_{o_0}^{b_0} \cdots \rightsquigarrow_{o_{n-1}}^{b_{n-1}} \mathcal{S}_n$, where $\mathcal{S}_0$ is an initial state in which only the main method is available. Note that from a given state there may be several possible execution steps that can be taken (since we have no assumptions on scheduling). In order to quantify the cost of an execution step, we use a cost model $\mathcal{M} : Ins \mapsto \mathbb{R}$ which maps instructions built using the grammar in Sec. 2.1 to real numbers. Below we discuss several cost models. The cost of an execution step is defined as $\mathcal{M}(\mathcal{S} \rightsquigarrow_o^b \mathcal{S}') = \mathcal{M}(b)$.

In the execution of sequential programs, the *cumulative cost of a trace* is obtained by applying a given cost model to each step of the trace. In our setting, this has to be extended because, rather than considering a single machine in which all steps are performed, we have a potentially distributed setting, with multiple objects possibly running concurrently on different CPUs. Thus, rather than aggregating the cost of all executing steps, it is more useful to treat execution steps which occur on different computing infrastructures separately. With this aim, we adopt the notion of *cost centers* [17], proposed for profiling functional programs. Since the concurrency unit of our language is the object, cost centers are used to

charge the cost of each step to the cost center associated to the object where the step is performed. For a given set of objects identifiers O and a trace t, we use $t|_O = \{\mathcal{S}_i \leadsto^{b_i}_{o_i} \mathcal{S}_{i+1} \mid \mathcal{S}_i \leadsto^{b_i}_{o_i} \mathcal{S}_{i+1} \in t,\ o_i \in O\}$ to denote the set of execution steps that are performed on objects from O. The cost of t w.r.t. a cost model $\mathcal{M}$ and a cost center O is $\mathcal{C}(t, O, \mathcal{M}) = \sum_{e \in t|_O} \mathcal{M}(e)$. Observe that it is also possible to apply different cost models to different cost centers.

Cost Models. We consider platform independent cost models (e.g., worst-case execution time is excluded). The following are cost models for approximating the *number of executed instructions* (left) and *memory consumption* (right):

$$\mathcal{M}_i(b) = \begin{cases} 0 & b \equiv \texttt{call(b, \_)} \\ \mathcal{M}_i(g_1) + \mathcal{M}_i(g_2) & b \equiv g_1 \wedge g_2 \\ 1 & \textit{otherwise} \end{cases} \qquad \mathcal{M}_m(b) = \begin{cases} |Co(\bar{t})| & b \equiv x := Co(\bar{t}), \\ 0 & \textit{otherwise} \end{cases}$$

$\mathcal{M}_i$ assigns cost 1 to all instructions except calls to blocks, as they do not appear in the original program. For memory consumption, $\mathcal{M}_m$ measures the size of constructed terms where $|x| = 0$ if x is a variable, $|Co(\bar{t})| = \mathit{size}(Co) + \Sigma_{t_i \in \bar{t}} |t_i|$, and $\mathit{size}(Co)$ denotes the memory required by the data constructor Co. Recall that objects are meant to be the concurrency units, while data structures are constructed using terms. A cost model that counts the *total number of objects created* along the execution can be define as $\mathcal{M}_o(b) = 1$ if $b \equiv$ `new` C and $\mathcal{M}_o(b) = 0$ otherwise. It provides an indication on the amount of parallelism that might be achieved. A cost model that counts `call(m, _)`, can be used to infer the *task-level* of the program [4], i.e., the number of tasks that are spawned along an execution. This can be used, for example, for finding optimal deployment configurations. We can also count the number of calls to specific methods or objects, e.g., by counting `call(m, _`$(o$`, _, _))` we obtain bounds on the number of requests to a remote component o. This is useful for approximating the components' load. Note that by '_' we mean any (valid) expression. The above cost models can also be used to prove termination of the program by setting the underlying solver [1] to only bound the number of iterations in loops.

4 The Basic Cost Analysis Framework

Our starting point is a powerful field-sensitive cost analysis framework for sequential OO programs [2, 3]. When lifting such framework to the concurrent setting, the main two difficulties and novelties are: (1) it is widely recognized that, due to the possible interleaving between tasks, tracking values of data stored in the heap is challenging. In Sec. 4.1, we present the basic, novel, *field-sensitive* size analysis for the concurrent setting, and in Sec. 5 we discuss how to further improve its precision; and (2) standard recurrence relations (in the sequential setting) assume a single cost center which accumulates the cost of the whole execution. In Sec. 4.2, we propose a novel form of *recurrence relations* which are parametric on the *cost centers* to which cost must be assigned.

4.1 Field-Sensitive Size Analysis for Concurrent OO Programs

The objective of size analysis is to infer *size relations* which allow reasoning on how the size of data changes along a program's execution, which is fundamental for bounding the number of iterations that loops perform. E.g., if a loop traverses a list, and we infer that the length of the list decreases at each iteration, then we can bound the number of iterations by the length of the list. We present the size analysis in three steps: recall the notion of size measure that maps data structures to their sizes; present an abstraction which compiles instructions into size constraints, trying to keep as much information on global data (i.e., fields) as possible, while still being sound in concurrent executions; and infer input-output size relations for increasing precision of inter-procedural analysis.

Size Measures. When a program manipulates terms, its cost usually depends on the *size* of the terms. E.g., the cost of traversing a list often depends on its length. We rely on the notion of *norms* [5] to define the *size of a term*. Norms are functions that map terms to their sizes. Any norm can be used in the analysis, depending on the nature of data structures used in the program. They can also be synthesized automatically from the program's type definitions. In what follows, w.l.o.g., we use the *term-size* norm, which counts the number of type constructors in a given term, defined as: $\left| Co(t_1, \ldots, t_n) \right|_{ts} = 1 + \Sigma_{i=1}^{n} \left| t_i \right|_{ts}$ and $\left| x \right|_{ts} = x$. Note that the size of a variable x is defined as x. In this way, we account for the size of the term to which x is bounded at runtime. For example, after executing $y = Cons(0, x)$, the size of x will be 2 plus the size of x. This is captured by abstracting this instruction to $y = 2 + x$.

In addition to terms and numerical values, our language includes reference variables (that point to objects) and future variables. In our context, objects are intended to simulate concurrent computing entities and not data structures, it is hence not common that they directly affect loop iterations. Therefore, ignoring their sizes is sound and precise enough in most cases. A slightly more precise abstraction distinguishes between the case in which a reference variable points to an object (size 1) or to null (size 0). The size of a future variable is the same as the size of the value it holds. This is sound since such variables can be used only through get, which blocks until the variable has a value.

Abstract Compilation. Modeling shared memory is a main challenge in static analysis of OO programs. Our starting point is [2], which models fields as local variables when the field to be tracked satisfies: (1) its memory location does not change; and (2) it is always accessed through the same reference (i.e., not through aliases). Both conditions can often be proven statically and the transformation of fields into local variables can then be applied for many fragments of the program. If we ignore concurrency, this approach could be directly adopted for our language. However, concurrency introduces new challenges.

Example 2. Consider the loop in the reqFile method in Fig. 2. Ignoring the `await` instruction, the above soundness conditions hold for field size, and hence, we can track it as if it were a local variable. In a concurrent setting, however, while reqFile is executing another task in the same object may modify size. Therefore, when

	b	$\alpha_\rho(b)$	ρ'
1	$e\ op\ e \quad op \in \{\wedge, >, \geq, =, +, -\}$	$\alpha_\rho(e)\ op\ \alpha_\rho(e)$	$\rho'{=}\rho$
2	$e\ op\ e \quad op \in \{*, /\}$	_	$\rho'{=}\rho$
3	t	$\lceil t[x \mapsto \rho(x)] \rceil_{ts}$	$\rho'{=}\rho$
4	**null** $\mid x \mid this.f \mid match(x,t)$	$0 \mid \rho(x) \mid \rho(f) \mid \rho(x){=}\alpha_\rho(t)$	$\rho'{=}\rho$
5	`release`	$true$	$\rho'{=}\rho[\bar{f}_C \mapsto \rho(\bar{f}_C)']$
6	`await` g	$\alpha_{\rho'}(g)$	$\rho'{=}\rho[\bar{f}_C \mapsto \rho(\bar{f}_C)']$
7	$x := y.$**get** $\mid x := e$	$\rho'(x){=}\rho(y) \mid \rho'(x){=}\alpha_\rho(e)$	$\rho'{=}\rho[x \mapsto \rho(x)']$
8	$this.f := e$	$\rho'(f){=}\alpha_\rho(e)$	$\rho'{=}\rho[f \mapsto \rho(f)']$
9	$x :=$ new C	$\rho'(x){=}1$	$\rho'{=}\rho[x \mapsto \rho(x)']$
10	`call`$(\mathtt{b}, q(rec, \bar{x}, \bar{y}))$	$q(\rho(rec), \rho(\bar{x}{\cdot}\bar{f}_C), \rho'(\bar{y}{\cdot}\bar{f}_C))$	$\rho'{=}\rho[\bar{y}{\cdot}\bar{f}_C{\mapsto}\rho(\bar{y}{\cdot}\bar{f}_C)']$
11	`call`$(\mathtt{m/f}, q(rec, \bar{x}, \bar{y}))$	$q(\rho(rec), \rho(\bar{x}), \rho'(\bar{y}))$	$\rho'{=}\rho[\bar{y} \mapsto \rho(\bar{y})']$
12	$otherwise$	$true$	$\rho'{=}\rho$

Fig. 4. Abstract compilation. $\mathrm{ABST}(b_{k:i}, \rho){=}\langle\alpha_\rho(b_{k:i}), \rho'\rangle$

analyzing reqFile, we cannot assume that the value of size is locally trackable. E.g, reqFile may not terminate if method void p() {size = size − 2; } is executing in parallel, since when `await` is executed inside the loop, p may change the value of size to a non-positive value, and thus the loop counter i may not decrement.

Handling fields requires identifying program points in which the shared memory might be modified by other tasks. This can happen when: (1) `release` or `await` are explicitly executed, and thus allow other tasks (of the same object) to run; and (2) an asynchronous invocation is issued, and until the called method starts to execute, the fields of the called object might be changed by other tasks. We refer to such program points as *release points*. The above observation suggests that in a sequence of instructions not including `release` or `await`, the shared memory can be tracked locally. However, the values in the shared memory when a method starts to execute may not be identical to those when it was called. We first present a safe abstraction which loses all information at release points and at method entries. In a second step we handle these points.

An abstract state is a set of linear constraints whose solutions define possible concrete states. This representation allows describing relations which are essential for inferring cost and proving termination, e.g., the size of x decreases by 1 in two consecutive states. The building blocks for this representation are constraints that describe the effect of each instruction b on a given state. We refer to such constraints as the *abstraction* of b. Fig. 4 depicts these abstractions. In order to abstract an instruction b, we use a mapping ρ from variables and field names to constraint variables that represent their sizes in the state before executing b. The result of abstracting b w.r.t ρ is the constraints $\alpha_\rho(b)$, and a new mapping ρ' that refers to the sizes in the state after executing b.

Let us describe the abstraction of some instructions. In Line 7, the instruction $x := e$ is abstracted into the equality $\rho'(x) = \alpha_\rho(e)$, where $\alpha_\rho(e)$ is the size of e w.r.t. ρ. E.g., if $e \equiv Cons(x, y)$, then Line 3 abstracts e to $1 + \rho(x) + \rho(y)$. Note that $\rho'(x)$ (resp. $\rho(x)$) refers to the size of x *after* (resp. *before*) executing the

instruction. The abstraction of `release` at Line 5 "forgets" sizes of the fields $\bar{f}_C$. This is because they might be updated by other methods that take the control when the current task suspends. The abstraction of `await` is similar, though we add to the abstract state the information that the guard g is satisfied upon completion of `await` g. When abstracting a call to a block in Line 10, the class fields are added as arguments in order to track their values. However, when abstracting calls to methods and functions (Line 11) the fields are not added. For methods, they are not added because their values at call time might not be the same as when the method actually starts to execute. For functions, they are not added since functions are not class members and cannot access fields. Since we use linear constraints only, non-linear arithmetic expressions (Line 2) are abstracted to a fresh constraint variable "_" that represents any value. A program P is transformed into an abstract program P^α, that approximates its behavior w.r.t. a size measure, by abstracting its rules as follows.

Definition 1 (Abstract Compilation). *Given* $r{\equiv}m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$, *and an identity map* ρ_0 *over* $vars(r) \cup \bar{f}_C$, *the abstract compilation of* r *is* $r^\alpha \equiv m(this, \bar{I}, \rho_{n+1}(\bar{O})) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$ *where:*

- $\langle g^\alpha, \rho_1\rangle{=}\mathit{ABST}(g, \rho_0)$, $\langle b_i^\alpha, \rho_{i+1}\rangle{=}\mathit{ABST}(b_i, \rho_i)$ *; and*
- $\bar{I}{=}\bar{x}{\cdot}\bar{f}_C$ *and* $\bar{O}{=}\rho_{n+1}(\bar{y}{\cdot}\bar{f}_C)$ *if* m *is a block; otherwise* $\bar{I}{=}\bar{x}$ *and* $\bar{O}{=}\rho_{n+1}(\bar{y})$.

Example 3. The following is the abstract compilation of *reqFile* of Fig. 3 where the tuple $\overline{F}$ denotes the fields $\langle db, file, catalog, myNeighbors, admin, size\rangle$:

	$reqFile(\langle this, s, fn\rangle, \langle\rangle) \leftarrow$	ρ_0
	$f' = 1,$	$\rho_1 = \rho_0[f \mapsto f']$
	$ps' = 1,$	$\rho_2 = \rho_1[ps \mapsto ps']$
	$i' = 0,$	$\rho_3 = \rho_2[i \mapsto i']$
	$incr' = 0$	$\rho_4 = \rho_3[incr \mapsto incr']$
③	$lengthNode(\langle s, fn\rangle, \langle l_1'\rangle),$	$\rho_5 = \rho_4[l_1 \mapsto l_1']$
①	$true,$	$\rho_6 = \rho_5[\overline{F} \mapsto \overline{F'}]$
	$i'' = l_1',$	$\rho_7 = \rho_6[i \mapsto i'']$
②	$loop(\langle this, s, fn, f', ps', i'', incr', l_2, \overline{F}'\rangle,$	$\rho_8 = \rho_7[f \mapsto f'', ps \mapsto ps'', l_2 \mapsto l_2',$
	$\langle f'', ps'', l_2', i''', incr'', \overline{F}''\rangle),$	$i \mapsto i''', incr \mapsto incr'', \overline{F} \mapsto \overline{F}'']$
	$thisDB' = db'',$	$\rho_9 = \rho_8[thisDB \mapsto thisDB']$
④	$storeFile(\langle thisDB', fn, f''\rangle, \langle\rangle).$	$\rho_{10} = \rho_9$

Note that at ① `await` is abstracted to *true* and the information on fields is lost, at ② the fields are added to the call in order to keep track of their values, however, when calling a method at ③ and ④, the abstraction "forgets" this information.

Input-Output Relations. There are relations between variables which cannot be observed at the rule level. E.g., at ③ above, the relation between the output variable l_1' of *lengthNode* and its input variables is not explicit, as it depends on the functionality of *lengthNode*. We refer to these relations as *input-output (size) relations*, IO relations for short. They describe post-conditions that hold, upon return, between the sizes of the input and output variables. They can be essential for bounding loops and the values to which an expression can be evaluated. The abstract program P^α can be used to infer such relations. For this, we use techniques developed for sequential programs [5], slightly modified in order to guarantee that non-terminating computations are approximated with

true rather than *false*, since in our setting method calls are asynchronous, and thus execution immediately returns to the calling site. In what follows, we assume that $\mathcal{I}_P$ is the set of such relations for all procedures. The elements of $\mathcal{I}_P$ take the form $\langle m(this, \bar{x}, \bar{y}), \psi \rangle$ where ψ is a set of constraints over $\bar{x} \cup \bar{y}$. If executing m on input of size $\bar{v}_1$ results in output of size $\bar{v}_2$, then $\bar{x}{=}\bar{v}_1 \wedge \bar{y}{=}\bar{v}_2 \models \psi$. We also require that ψ does not restrict the input, i.e., $\forall \bar{v}_1 \in \mathbb{Z}. \exists \bar{v}_2 \in \mathbb{Z}. \bar{x}{=}\bar{v}_1 \wedge \bar{y}{=}\bar{v}_2 \models \psi$.

Example 4. In method lengthDB in Fig. 2, the output of *lookup* is an input to *length*. Thus, in order to infer the cost of lengthDB, we need IO relations for *lookup*. Using the techniques of [5], we infer that the size of *lp* is smaller than that of *ms*, i.e., $\langle lookup(\langle this, ms, k \rangle, \langle lp \rangle), \{2{+}lp{+}k{\leq}ms\} \rangle \in \mathcal{I}_P$.

4.2 Cost Relations Based on Cost Centers

The next step is to generate *Cost Relation Systems* (CRS) which define the cost of executing each method as a function of its parameters and the initial state of the fields when the method starts to execute. The main novelty is that CRS use cost centers to track the resource usage of the different components separately. Given a finite set of cost centers $c_0, \ldots, c_n$, where c_0 denotes the cost center of the main method, we assume the existence of a function $\texttt{CC}(o)$ which returns statically a set of possible cost centers of object o at a given program point. This allows instantiating our analysis with different deployment strategies. In particular, when a group of objects share the processor, as in JCobox [18], then they belong to the same cost center. In the examples, for simplicity, we assume that objects of the same class belong to the same cost center, i.e., $\texttt{CC}(o)$ returns the class of o. For this case, CC is computed automatically using class analysis.

Definition 2 (CRS). *Let* $r \equiv m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$, *and* $r^\alpha \equiv m(this, \bar{I}, \bar{O}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha \in P^\alpha$. *The cost equation of* r *w.r.t. a cost model* $\mathcal{M}$ *is* $m(D_0, \bar{I}) = e + q_1(D_1, \bar{x}_1) + \ldots + q_m(D_m, \bar{x}_m), \varphi$ *where:*
1. $e = \texttt{c}(D_0) * \mathcal{M}(g) + \Sigma_{i=1}^{n} \texttt{c}(D_0) * \mathcal{M}(b_i)$;
2. $\varphi {\equiv} g^\alpha \wedge (\bigwedge_{i=1}^{n} \varphi_i)$, *s.t. if* $b_i^\alpha \equiv q(rec, \bar{x}, \bar{y})$ *then* $\langle b_i^\alpha, \varphi_i \rangle \in \mathcal{I}_P$; *otherwise* $\varphi_i {\equiv} b_i^\alpha$;
3. each $q_i(rec, \bar{x}_i, \bar{y}_i) {\in} r^\alpha$ *defines* $q_i(D_i, \bar{x}_i)$ *s.t.* $D_i{=}D_0$ *if* $rec{\equiv}this$, *else* $D_i{=}\texttt{CC}(rec)$.

The CRS of P, denoted P^{crs}, is the set of cost equations of its rules. The generated equations are similar to those for sequential programs [4] in that: (i) the cost is a function of the input; (ii) they are obtained by applying the cost model $\mathcal{M}$ to each instruction in the body of r (point 1); (iii) the size relations gathered in φ define the applicability constraints (point 2); and (iv) a call in the program induces a call in the CR (point 3). The generated equations are different from those of sequential programs in that: (a) they contain a cost center parameter D_0 representing the set of cost centers to which the cost will be assigned. The cost of calls (point 3) is assigned to the cost centers of the calling object; (b) the cost expressions we accumulate (point 1) are multiplied by $\texttt{c}(D)$, which allow us to obtain the cost for a particular cost center c_i by setting $\texttt{c}(D)$ to 1 if $c_i \in D$, and to 0 otherwise. Given a cost expression e, we let $e|_{D'}$ be the result of replacing $\texttt{c}(D)$ by 1 if $D \cap D' \neq \emptyset$ and 0 otherwise. Note that since the generated

CRS does not include concurrency constructs, it can be solved to closed-form UBs/LBs using existing solvers [1] developed for the sequential setting.

Example 5. The following are selected equations from the CRS resulting from applying Def. 2 to *reqFile* and its reachable methods w.r.t. $\mathcal{M}_i$, assuming that $\mathtt{CC}(o)$ returns the class of o. For readability, we have removed all arguments and intermediate constraints that do not affect the cost and implicitly assumed that $x \geq 1$ for any variable x containing a term (i.e., its size is positive). The first parameter in the equations is instantiated to the cost center to which the cost will be assigned. When a method call on an object is performed (annotated as ①), the cost center gets instantiated with the actual class.

$$
\begin{array}{rl}
\mathit{reqFile}(\mathsf{Node}, s, \mathit{fn}) = & \mathsf{c}(\mathsf{Node}) * 15 + ①\mathit{lengthNode}(\mathsf{Node}, \mathit{fn}) + \\
& \mathit{loop}(\mathsf{Node}, \mathit{fn}, i'', \mathit{size}) + ①\mathit{storeFile}(\mathsf{DB}) \quad \{i'' \geq 0\} \\
\mathit{loop}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) = & \mathsf{c}(\mathsf{Node}) * 3 \quad \{i \leq 0\} \\
\mathit{loop}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) = & \mathsf{c}(\mathsf{Node}) * 4 + \mathit{if}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) \quad \{i > 0\} \\
\mathit{if}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) = & \mathsf{c}(\mathsf{Node}) * 3 + \mathit{if}^c(\mathsf{Node}, \mathit{fn}, i, \mathit{incr}, \mathit{size}) \ \{i < \mathit{size}, \mathit{incr} = i\} \\
\mathit{if}(\mathsf{Node}, \mathit{fn}, i, \mathit{size}) = & \mathsf{c}(\mathsf{Node}) * 3 + \mathit{if}^c(\mathsf{Node}, \mathit{fn}, i, \mathit{incr}, \mathit{size}) \ \{i \geq \mathit{size}, \mathit{incr} = \mathit{size}\} \\
\mathit{if}^c(\mathsf{Node}, \mathit{fn}, i, \mathit{incr}, \mathit{size}) = & \mathsf{c}(\mathsf{Node}) * 10 + ①\mathit{getPacks}(\mathsf{Node}, \mathit{fn}, \mathit{incr}, i) + \\
& \mathit{app}(\mathsf{Node}, \mathit{ps}, f) + \mathit{loop}(\mathsf{Node}, \mathit{fn}, i', \mathit{size}') \\
& \{\ \mathit{ps} + f - 1 = f',\ i' = i - \mathit{incr}\} \\
②\mathit{lengthDB}(\mathsf{DB}, \mathit{fn}) = & \mathsf{c}(\mathsf{DB}) * 2 + \mathit{lookup}(\mathsf{DB}, \mathit{dbf}, \mathit{fn}) + \mathit{length}(\mathsf{DB}, \mathit{lp}) \\
& \{\mathit{lp} \geq 1, 3 + \mathit{lp} + \mathit{fn} \leq \mathit{dbf}\}
\end{array}
$$

The first equation defines the cost *reqFile* in terms of those of *lengthNode*, *loop* and *storeFile*. The expression $\mathsf{c}(\mathsf{Node}) * 15$ corresponds to the cost of the instructions in reqFile which are outside the loop. The second (resp. third) equation corresponds to the case when $i \leq 0$ (resp. $i > 0$) of the while loop condition. The equations of *if* correspond to the then ($i < \mathit{size}$) and else ($i \geq \mathit{size}$) branches where *incr* is assigned to i or *size* and it continues to if^c. The equation if^c corresponds to the loop's body, it accumulates the cost of *getPacks* and *app*, and recursively calls *loop*. Note that i is decremented by *incr* units, and that the value of the field *size* is lost (we use size') due to the await. The accuracy of the analysis shows up when solving the CRS into closed-forms. None of the above equations is solvable. Using [1], we only obtain UBs for functions, e.g.:

$$
\begin{array}{rl}
\mathit{length}(D_0, \mathit{list}) = & \mathsf{c}(D_0) * 3 + \mathsf{nat}(\frac{\mathit{list}-1}{2}) * \mathsf{c}(D_0) * 6 \\
\mathit{lookup}(D_0, \mathit{ms}, k) = & \mathsf{c}(D_0) * 2 + \mathsf{nat}(\frac{\mathit{ms}-1}{2}) * \mathsf{c}(D_0) * 4
\end{array}
$$

The failure in solving the CRS of all methods is due to the loss of information in the abstract compilation. Let us focus on *lengthDB*. Substituting the above UBs in equation ② results in (with the size relation as above):

$$\mathit{lengthDB}(\mathsf{DB}, \mathit{fn}) = \mathsf{c}(\mathsf{DB}) * 7 + \mathsf{nat}(\tfrac{\mathit{dbf}-1}{2}) * \mathsf{c}(\mathsf{DB}) * 4 + \mathsf{nat}(\tfrac{l-1}{2}) * \mathsf{c}(\mathsf{DB}) * 6$$

The problem is that we cannot bound the value of *dbf*, since information on fields has been lost. Therefore, we cannot find the maximal cost of *lengthDB*.

The next theorem guarantees that an UB (resp. LB) of the CRS is a correct UB (resp. LB) on the actual cost. Given an equation m and a set of cost centers D, $\mathit{Ans}(m(D, \bar{v}))$ denotes all possible answers for $m(D, \bar{v})$ in the corresponding CR [1]. The elements of $\mathit{Ans}(m(D, \bar{v}))$ are symbolic expressions over $\mathsf{c}(D)$.

$$lengthDB(\mathsf{DB}, fn) = \mathsf{c(DB)}{*}(7{+}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}){*}4{+}\mathsf{nat}(\tfrac{dbf_{max}-fn}{2}-2){*}6)$$

$$getFile(\mathsf{DB}, fn) = \mathsf{c(DB)}{*}(3{+}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}){*}4)$$

$$lengthNode(\mathsf{Node}, fn) = \mathsf{c(Node)}{*}4{+}\mathsf{c(DB)}{*}(7{+}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}){*}4{+}\mathsf{nat}(\tfrac{dbf_{max}-fn}{2}-2){*}6)$$

$$getPacks(\mathsf{Node}, fn, ps, n) = \mathsf{c(Node)}{*}13{+}\ \mathsf{c(DB)}{*}(3{+}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}){*}4){+}\mathsf{c(Node)}{*}\mathsf{nat}(ps){*}(18{+}\mathsf{nat}(ps{+}n{-}1){*}9)$$

$$reqFile(\mathsf{Node}, s, fn) = \ \textcircled{a}\ \mathsf{c(Node)}{*}15{+}$$
$$\textcircled{b}\ \ \mathsf{c(Node)}{*}4{+}\mathsf{c(DB)}{*}(7{+}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}){*}4{+}\mathsf{nat}(\tfrac{dbf_{max}-fn}{2}-2){*}6){+}$$
$$\textcircled{c}\ \ \mathsf{nat}(\tfrac{dbf_{max}-fn}{2}-2){*}($$
$$\textcircled{d}\ 17{*}\mathsf{c(Node)}{+}$$
$$\textcircled{e}\ \begin{cases}\mathsf{c(Node)}{*}13{+}\mathsf{c(DB)}{*}(3{+}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}){*}4){+}\\ \mathsf{c(Node)}{*}\mathsf{nat}(size_{init}){*}(18{+}\mathsf{nat}(\tfrac{dbf_{max}-fn}{2}-3)){*}9{+}\end{cases}$$
$$\textcircled{f}\ \ \mathsf{c(Node)}{*}(2{+}5{*}\mathsf{nat}(\tfrac{dbf_{max}-1}{2}))){+}\ \textcircled{g}\ \mathsf{c(DB)}{*}3$$

Fig. 5. Upper Bounds for Selected Methods using Class Invariants

Theorem 1 (Soundness). *Given a program P, a cost model $\mathcal{M}$, and a set of cost centers D. Then for any trace t starting from an initial configuration, there exists $e \in Ans(\texttt{main}(\{c_0\}))$ such that $e|_D = \mathcal{C}(t, D, \mathcal{M})$.*

5 Class Invariants in Cost Analysis

In this section, we propose a generalization of *class invariants* (see, e.g., [16]) which allows highly improving the accuracy of the size analysis in Sec. 4.1. As discussed in Sec. 4, release points are problematic since at these points other task(s) may modify the values of shared fields. However, it is often possible to gather useful information about shared variables, in the form of class invariants, which must hold at those points. In sequential programs, class invariants have to be established by constructors and must hold on termination of all (public) methods of the class. They can be assumed at (public) method entry but may not hold temporarily at intermediate states not visible outside the object. In our context, we need that such invariants hold on method termination and also at all release points of all methods. This way, we can use them to improve the abstraction at the release points. In the following, given a class C, Ψ_C denotes the class invariant for class C, which is a set of linear constraints over the fields of C and possibly some constant symbols.

Definition 3. *We extend Def. 1 as follow: (1) when abstracting a method rule, we add Ψ_C to the abstract rule (just before g^α); and (2) we abstract* `release` *(resp.* `await`*) to $\Psi_C[\bar{f}_C \mapsto \rho'(\bar{f}_C)]$ (resp. $\alpha_\rho(g) \wedge \Psi_C[\bar{f}_C \mapsto \rho'(\bar{f}_C)]$).*

Example 6. The following invariants are required to solve the equations in Ex. 5: (1) In class DB, we need an invariant $0 \leq dbf \leq dbf_{max}$, where dbf_{max} is a constant symbol which bounds the value of dbf; and (2) In class Node, we need an invariant which establishes that $size = size_{init}$, i.e., field size is initialized in the constructor and it is never modified. By applying Def. 3 using the first

invariant, the equations for *lengthDB* is like that annotated as ② in Ex. 5 but including the additional constraint $\{0 \leq dbf \leq dbf_{max}\}$. This allows obtaining the UBs in Fig. 5 for *lengthDB*, as well as for *getFile*, *lengthNode* and *getPacks*, whose costs depend on them. The second invariant is essential in order to obtain an UB for *reqFile*. In particular, it is needed in the equation if^c in Ex. 5, which corresponds to the cost of the block that contains the `await` instruction (and that introduced inaccuracy in the analysis). By applying Def. 3 using such invariant, the solver [1] obtains the UB for *reqFile* in Fig. 5. Let us explain the different parts of this UB: ⓐ is the cost of the instructions of *reqFile* excluding those of the loop; ⓑ is the cost introduced by *lengthNode*; ⓒ is the number of iterations of the loop; ⓓ − ⓕ is the cost of each iteration of the loop, where ⓓ is the cost of the loop's instructions, ⓔ is the cost of calls to *getPacks*, and ⓕ is the cost of calls to *app*; and ⓖ is the cost of *storeFile*. As expected, the number of executed instructions has an asymptotic bound $O(dbf_{max}^2 * size_{init})$. The cost on the cost center Node is $O(dbf_{max}^2 * size_{init})$ while that on DB is $O(dbf_{max}^2)$. For *getPacks* the cost on the cost center DB is $O(dbf_{max})$ while on Node it is $O(ps^2)$.

6 Experiments: The COSTABS System

We have developed COSTABS, a cost analyzer of ABS programs. It uses [1] for solving the resulting CR. The system can be tried out online at: `http://costa.ls.fi.upm.es/costabs`. Experimental evaluation has been carried out using several typical concurrent applications: PeerToPeer, our running example; BookShop, a web shop client-server application; BBuffer, a classical bounded-buffer for communicating several producers and consumers; DistHT, a distributed hash-table, and PingPong, a simple communication protocol.

Table 1 summarizes our experiments. They have been performed on an Intel Core i5 at 3.2GHz with 3.1GB of RAM, running Linux. Each program is analyzed for proving termination and for obtaining an UB on the number of executed instructions ($\mathcal{M}_i$). For each benchmark, columns #**F** and #**M** show, resp., the number of functions and methods that have been analyzed. Regarding functions, we have proved termination and found UBs of all of them, without using class invariants. Column #$\mathbf{M}_t$ (resp. #$\mathbf{M}_{ub}$) show the number of methods for which COSTABS proves termination (resp. finds UBs), without using invariants. Column #$\mathbf{I}_t$ shows the number of class invariants needed to prove

Table 1. Statistics about the Analysis Process (times are in milliseconds)

Bench	#**F**	#**M**	#$\mathbf{M}_t$	#$\mathbf{M}_{ub}$	#$\mathbf{I}_t$	#$\mathbf{I}_{ub}$	$\mathbf{T}_{ir}$	$\mathbf{T}_{ac}$	$\mathbf{T}_{io}$	$\mathbf{T}_{term}$	$\mathbf{T}_{ub}$
P2P	11	17	14	6	1	2	9	17	69	1494	1886
BookShop	28	11	11	6	0	3	11	18	66	802	1520
BoundedBuffer	4	8	8	4	0	1	1	2	7	70	86
DistHT	7	8	8	1	0	2	3	5	12	82	100
PingPong	0	6	6	6	0	0	3	3	7	22	22

termination. Note that only one invariant (number 2 of Ex. 6) is required. Similarly, $\#\mathbf{I}_{ub}$ is the number of class invariants needed to find UBs. They are all similar to number 1 in Ex. 6. Note that the whole process is fully automatic.

Columns $\mathbf{T}_{ir}$, $\mathbf{T}_{ac}$, and $\mathbf{T}_{io}$ show, resp., the times taken to build the intermediate representation, abstract compilation and inference of IO relations. Columns $\mathbf{T}_{term}$ and $\mathbf{T}_{ub}$ show, resp., the times taken by [1] to solve the CRS for proving termination and for obtaining an UB w.r.t. $\mathcal{M}_i$. Proving termination requires less work than finding UBs, thus, $\mathbf{T}_{term} < \mathbf{T}_{ub}$. Although the anlyzer is still prototypical, we argue that the experiments show that our approach is promising.

7 Conclusions and Related Work

We have presented, to the best of our knowledge, the first cost analysis framework for concurrent programs. To develop the analysis, we have considered an OO language based on the notion of concurrent objects which live in a distributed environment with asynchronous communication. Most of our techniques would be also applicable to other concurrent programming languages. In particular, the idea of having equations parametric on the cost centers is of general applicability. The size analysis is tailored for the concurrency primitives of our language, but similar abstractions could be developed for other languages by finding correspondences between their concurrency primitives.

Our work is closely related to other resource usage analysis frameworks [11, 12]. Most of such frameworks assume a sequential execution model and thus do not deal with the main challenges addressed in this paper. Notable exceptions are [10, 15]. In [15], a live heap space analysis for a concurrent language is proposed, but for a simple model of shared memory which only considers a particular type of resource (memory). [10] introduce *dynamic matrices* for modeling cost analysis of concurrent programs. The use of cost centers has been proposed in the context of profiling, but to our knowledge, its use in the context of static analysis is new. The termination of multi-threaded programs presented in [6], is based on inferring conditions on the global state which are sufficient to guarantee termination and are similar to our class invariants. Observe that such conditions are only one component within our cost analysis framework, which additionally requires the generation of a new form of recurrece relations and the definion of cost models for the concurrent setting. When considering cumulative cost models, as we do in this paper, asynchronous calls can be handled exactly as synchronous calls without sacrifying precision. This is because, in such cost models, what is important is to approximate the number of times a method is executed (i.e., called), and not how many of them might be running in parallel. In contrast, when considering noncumulative cost models, information on the lifetime of each task is important, since it might directly affect the peak consumption of the corresponding resource. As future work, we plan to integrate in our framework cost models that are noncumulative [4].

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the EC, 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 UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Ramírez Deantes, D.V.: From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 100–116. Springer, Heidelberg (2010)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
4. Albert, E., Arenas, P., Genaim, S., Zanardini, D.: Task-Level Analysis for a Language with Async-Finish parallelism. In: Proc. of LCTES 2011, pp. 21–30. ACM (2011)
5. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
6. Cook, B., Podelski, A., Rybalchenko, A.: Proving Thread Termination. In: Proc. of PLDI 2007, pp. 320–330. ACM (2007)
7. Crary, K., Weirich, S.: Resource Bound Certification. In: Proc. of POPL 2000, pp. 184–198. ACM (2000)
8. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
9. Fähndrich, M.: Static Verification for Code Contracts. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 2–5. Springer, Heidelberg (2010)
10. Ferrari, G.L., Montanari, U.: Dynamic Matrices and the Cost Analysis of Concurrent Programs. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 307–321. Springer, Heidelberg (1995)
11. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: Proc. of POPL 2009, pp. 127–139. ACM (2009)
12. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010)
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Proc. FMCO 2010. Springer, Heidelberg (2010)
14. Johnsen, E.B., Owe, O.: An Asynchronous Communication Model for Distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
15. Kero, M., Pietrzak, P., Nordlander, J.: Live Heap Space Bounds for Real-Time Systems. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 287–303. Springer, Heidelberg (2010)

16. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (1997)
17. Jarvis, S.A., Morgan, R.G.: Profiling Large-Scale Lazy Functional Programs. Journal of Functional Programing 8(3), 201–237 (1998)
18. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
19. Spoto, F., Mesnard, F., Payet, É.: A Termination Analyser for Java Bytecode based on Path-Length. ACM TOPLAS 32(3) (2010)
20. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
21. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: Proc. of CASCON 1999, pp. 125–135. IBM (1999)
22. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM 18(9) (1975)

Static Object Race Detection

Ana Milanova and Wei Huang

Rensselaer Polytechnic Institute

Abstract. We present a novel static object race detection analysis. Our analysis is data-centric in the sense that dominance and ownership, as well as object-based reasoning about control, play a crucial role. Our empirical results show that the analysis scales well and has relatively low false-positive rate. In some cases, our analysis outperforms the leading static race detector Chord.

1 Introduction

A multithreaded program contains an object race when two threads invoke methods on the same object "simultaneously" (i.e., without ordering constraints between them). An object race is a generalization of a data race [15]. It may or may not lead to a data race; however, an object race is necessary in order for a data race to occur. Reasoning about object races is valuable in several ways. First, it entails reasoning about object structure, in particular dominance-based ownership structure [3], which may facilitate localization and correction of concurrency bugs. Second, it complements data race detection because object races may expose hidden data races (e.g., data races on internal objects of library classes, which typically are not reported by data race detectors).

In this paper we present a novel static analysis for object race detection. Dominance, as well as object-based reasoning about control, play a crucial role.

Dominance is defined in terms of the notion of *object graph*. Nodes in the object graph are objects, and edges capture references between those objects. An edge links object i to object j if i has a field that refers to j, or a variable in a method invoked on receiver i, refers to j. Object i dominates (or owns) j if all paths from the root of the object graph to j go through i. Dominance plays an important role in object race detection. Namely, synchronization on a dominator i protects all objects j internal to i's dominance boundary. Conversely, lack of synchronization on i may expose object races deep in the boundary of i. Fig. 1 shows a program and Fig. 2 shows an *abstract object graph* for this program. In this example, allocation sites j, m, b, t and w are executed many times, resulting in many concrete objects. A typical static abstraction scheme maps every concrete object to its allocation site, thus these concrete objects are mapped to the same abstract objects j, m, b, t, and w.

Additionally, we define the notion of the *call graph*. Nodes in the call graph are tuples $i.\mathsf{m}$ where i is an object and m is a method name; the tuple denotes that method m executes on receiver object i. The edges represent calls: there is an edge from $i.\mathsf{m}$ to $j.\mathsf{n}$ if method m executing on receiver i calls method

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 255–271, 2011.

n on receiver j. This object-based call graph is natural for object-oriented languages where objects and control are inherently intertwined and synchronization is naturally object-based. It facilitates object race detection. For example, when control descends into i.m, if m holds the lock on i, this lock protects not only i but all objects j dominated by i, that are accessed along the call chain from i.m. Fig. 2 shows the *abstract call graph* for the example program.

```
class J extends Thread {
  static C c; int wId;
  static void main(String[] arg) {
   c = new C();   [c]
   for (int num=1; num<=3; num++) {
    c.inc();
    for (int wID=0; wID<num; wID++) {
      J j = new J();   [j]
      j.wId = wID;
      j.start();
    }
   }
  }
  public void run() {
    M m = new M();   [m]
    m.init(c,whId); c.addThread(m);
    m.go();
  }
}

class C {
  W[] a = new W[10];   [a]
  int num = 0;
  synchronized void inc() {
    W w = new W();   [w]
    a[num++] = w;
   }
  synchronized void addThread(M m) { ... }
  synchronized W getW(int i) {
    W w = a[i]; return w;
  }
}

class W {
  int count = 0; S s = ...;
  void update(H h) {
    this.count++; s.put(h);
  }
  synchronized get() {
    return s;
  }
 }

class M {
   int wId; C c;
   T[] b = new T[10];   [b]
   void init(int wID, C c) {
     this.wId = wID; this.c = c;
   }
   void go() {
     T t = new T()   [t]
     b[0] = t; t.init(wId,c); t.process();
   }
}

class T {
  int wId; C c; W w;
  void init(int wID, C c) {
    wId = wID; this.c = c;
    w = c.getW(wId);
  }
  void process() {
    S s = w.get(); ... w.update(new ...);
  }
}
```

Fig. 1. Example program

Our analysis classifies objects as *distributed* or *owned*. A distributed object is dominated only by the root of the object graph. In contrast, an owned object is dominated by at least one object (owner). The analysis first identifies races on distributed objects, and then descends into the dominance boundary of each object to identify races on owned objects. The main intuition is that in order

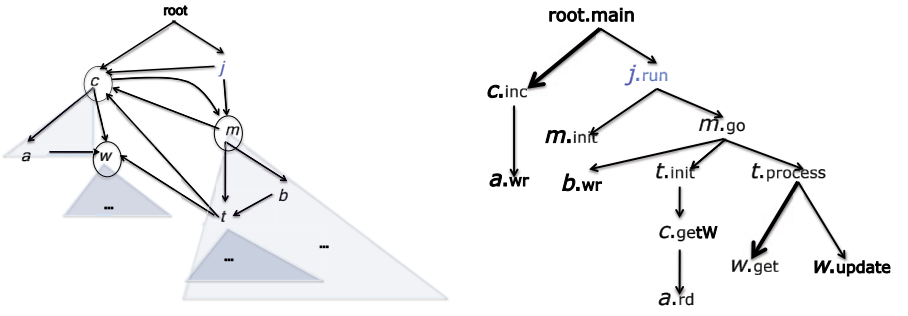


Fig. 2. Abstract object and call graphs for example program

to have a race on an owned object, we must first have a race on its dominator (owner). The analysis is best illustrated by an example. Consider Fig. 1 (modeled after benchmark SPECjbb). Method main forks multiple threads that act on the C (Company) object stored in static field c. Each thread creates a M (TransactionManager) object which in turn creates multiple T (Transaction) objects each accessing the C object and the W (Warehouse) objects. Our analysis identifies objects w, c and m as distributed (they are circled in the object graph in Fig. 2). There are two object races on w: $\langle w, \mathsf{get}, \mathsf{update}\rangle$ and $\langle w, \mathsf{update}, \mathsf{update}\rangle$. There are no races on c because all accesses to c are synchronized, and there are no races on m because each m is accessed only by its creating thread. The analysis proceeds to identify races in the boundary of w that are triggered by the two object races on w, $\langle w, \mathsf{get}, \mathsf{update}\rangle$ and $\langle w, \mathsf{update}, \mathsf{update}\rangle$. The lack of races on c entails that there are no races on a — a is owned by c and all accesses to a are protected by the lock on owner c. The lack of races on m entails that there are no races on owned t and b.

We have implemented the analysis and present results on several benchmarks. Our analysis presents relatively low false-positive rate and runs in less than 1 minute on all but one benchmark. On most benchmarks our analysis performs comparably to Chord [9], the leading static data race detector. On several benchmarks our analysis outperforms Chord, in some cases significantly.

The rest of the paper proceeds as follows. Section 2 formalizes the notions of object graph and call graph and presents a static analysis (abstract interpretation [4]) that infers safe abstract object and call graph. Section 3 describes the dominance inference analysis. Section 4 presents the object race detection analysis. Section 5 describes our implementation and experience with the analysis, Section 6 discusses related work and Section 7 concludes the paper.

2 Formal Account of Object Graphs

We explain our algorithm in terms of a core Java-like calculus. Throughout the paper we will use the following notation for graphs. An *object graph* G is a pair

$$
\begin{array}{lll}
cd & ::= \mathsf{class\ C\ extends\ D}\ \{\overline{fd}\ \overline{md}\} & \textit{class} \\
fd & ::= \tau\ \mathsf{f} & \textit{field} \\
md & ::= \tau\ \mathsf{m}(\overline{\tau\ \mathsf{x}})\{\overline{\tau\ \mathsf{z}};\ \mathsf{return\ y}\} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s;s} \mid \mathsf{x = new\ C()} \mid \mathsf{x = this.f} & \textit{statement} \\
 & \quad \mid \mathsf{this.f = y} \mid \mathsf{x = y.m}(\overline{\mathsf{z}}) & \\
\tau & ::= \mathsf{C} & \textit{type}
\end{array}
\qquad
\begin{array}{lll}
T & ::= tS & \textit{thread} \\
H & ::= [] \mid H[i \mapsto o] & \textit{heap} \\
S & ::= \epsilon \mid \langle \mathsf{m}\ F\ \mathsf{s} \rangle S & \textit{stack} \\
F & ::= [] \mid F[\mathsf{y} \mapsto i] & \textit{frame} \\
o & ::= \mathsf{C}(\overline{i}) & \textit{object}
\end{array}
$$

Fig. 3. Syntax

(N, E) where N is a set of objects ranged over by variables i, j, k, l and E is a set of directed edges written $i \triangleright j$. We write $i \in G$ and $i \triangleright j \in G$ to test, respectively node and edge membership. A *call graph* C is a pair (N, E) where N is a set of tuples written $i.\mathsf{m}$, where i is an object and m is a method, and E is a set of directed edges written $i.\mathsf{m} \triangleright j.\mathsf{n}$. The meaning of a call graph edge is that method m invoked on receiver object i calls method n on receiver object j. Again, we write $i.\mathsf{m} \in C$ and $i.\mathsf{m} \triangleright j.\mathsf{n} \in C$ to test membership.

2.1 Concrete Semantics

For brevity, we restrict our formal attention to a core calculus in the style of [14] whose syntax appears in Fig. 3. The language models Java with a syntax in A-normal form. Fields are strongly private. Array accesses are modeled by special methods rd and wr — array read x=y[i] is treated as method call x = y.rd() and array write x[i]=y is treated as x.wr(y); index i is irrelevant for our purposes and is omitted. Throughout the paper, metavariables m and n range over all method names and rd and wr. Features not strictly necessary are omitted.

The concrete semantics operates over configurations of the form $H; \overline{T}; G; C; P$ where H is a single heap, $\overline{T}$ is a collection of threads, G is a summary object graph, C is a summary call graph and P holds auxiliary information necessary to construct C. A heap is a mapping from indices, ranged over by meta-variables i, j, k, l, to objects. Each thread T has its own stack S and a unique thread identifier t. A stack is a sequence of frames $\langle \mathsf{m}\ F\ \mathsf{s} \rangle$ consisting of a method name m, a mapping F from variables to locations and a statement s. An object $o = \mathsf{C}(\overline{i})$ consists of a class C and values $\overline{i}$ for the object fields. An object graph G summarizes all references between objects. A call graph C summarizes all method calls between objects.

We write $\overline{i}$ to denote a sequence of indices, $\overline{\tau\ \mathsf{z}}$ for a sequence of local variable declarations, etc. We write 0 to denote the null reference.

Following [14], a multi-threaded Java program is modeled as a fixed set of threads $\overline{T}$, each of which starts with a call to a run method, and terminates when the run method returns. The reduction relation $\xrightarrow{l}_t$ represents a step in the semantics; l is an action label and t is the identifier of the thread that executed that action. We use action labels on methods calls $\rightarrow i.\mathsf{m}$ (call), and on method returns $\leftarrow i.\mathsf{m}$ (return), as well as the empty label ϵ. Later in the

paper, labels are used to define traces and object races. Thread scheduling is modeled as a non-deterministic choice where each step picks one of the threads for reduction (see [14] for the rule). Due to space constraints, the rules of the concrete semantics are not shown here.

2.2 Abstract Semantics

We assume a *may* points-to analysis that computes a safe approximation of the heap $\widehat{H}$, collection of threads $\widehat{\overline{T}}$, and each stack $\widehat{S}$. The abstract semantics computes safe approximations of G and C, denoted $\widehat{G}$ and $\widehat{C}$ respectively. As $\widehat{H}$ and $\widehat{S}$ are conservative approximations, the semantics operates on sets of *abstract objects*. Thus, $\widehat{F}(\mathsf{x})$ evaluates to a set of abstract objects, not to a single object. Similarly, fields of an object in $\widehat{H}$ are sets of references (denoted I). We assume that all allocation sites are labelled with an unique identifier.

The abstraction function α is specific to our points-to analysis and is chosen so that $\alpha(i) = i'$ where i' is the index of the allocation site that created i. The abstraction applies to threads as well: $\alpha(t) = t'$ where t' is the index of the allocation site that created the $\mathsf{java.lang.Thread}$ object that started t's run. α acts on G in the obvious way: $\alpha(G) = (N, E)$, where $N = \{\alpha(i) \mid i \in G\}$ and $E = \{\alpha(i) \triangleright \alpha(j) \mid i \triangleright j \in G\}$. Similarly, α acts on C: $\alpha(C) = (N, E)$ where $N = \{\alpha(i).\mathsf{m} \mid i.\mathsf{m} \in C\}$ and $E = \{\alpha(i).\mathsf{m} \triangleright \alpha(j).\mathsf{n} \mid i.\mathsf{m} \triangleright j.\mathsf{n} \in C\}$.

As the points-to analysis is safe, the following two conditions hold at every step. The first condition ensures the safety of variables, and the second ensures the safety of fields.

$$\begin{array}{rcl} F(\mathsf{x}) = i & \Rightarrow & \alpha(i) \in \widehat{F}(\mathsf{x}) \\ H(i) = \mathsf{C}(...k_{\mathsf{f}}...) & \Rightarrow & \widehat{H}(\alpha(i)) = \mathsf{C}(...I_{\mathsf{f}}...) \ \wedge \ \alpha(k_{\mathsf{f}}) \in I_{\mathsf{f}} \end{array}$$

The rules of the abstract semantics use $\widehat{H}$ and $\widehat{F}$ and compute $\widehat{G}$ and $\widehat{C}$. We write $\widehat{G}$ += $i \triangleright j$ to denote the addition i and j to the nodes of $\widehat{G}$ and $i \triangleright j$ to the edges of $\widehat{G}$. Similarly, we write $\widehat{C}$ += $i.\mathsf{m} \triangleright j.\mathsf{n}$ to denote the addition of $i.\mathsf{m}$ and $j.\mathsf{n}$ to the nodes of $\widehat{C}$, and $i.\mathsf{m} \triangleright j.\mathsf{n}$ to the edges of $\widehat{C}$. Auxiliary function *dispatch* takes as argument the class of the receiver C and the call site id c and returns the run-time target n.

Fig. 4(a) (left column) shows the rules for constructing object graph $\widehat{G}$. (ANEW) adds new edges to $\widehat{G}$ from every abstract receiver i of current frame m, to the abstract object j created at allocation site j. Rule (ACALL) adds new edges to $\widehat{G}$ from every abstract object k in the points-to set of y, to every j in the points-to set of an actual argument z, and from each abstract receiver i of method m, to each j in the points-to set of the return variable of n, $\mathsf{ret_n}$. Note that calls through this, e.g., $\mathsf{this.n}(\overline{\mathsf{z}})$, do not add edges to $\widehat{G}$. This is correct because when the call is through this the relevant abstract edges are already in $\widehat{G}$ and there is no need to add them again.

Fig. 4(b) (right column) shows the rules for constructing call graph $\widehat{C}$. The first two rules compute sets $\widehat{P}_{k.\mathsf{n}}$, the set of caller tuples for $k.\mathsf{n}$. (ACALL) adds

(ANEW)
$\mathsf{x} = \mathsf{new}_j\ \mathsf{C}()$ in method $\mathsf{m} \Rightarrow$
$\quad \widehat{G}$ += $\{k \triangleright j | k \in \widehat{F}(\mathsf{this_m})\}$

(ACALL)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = dispatch(\mathsf{C}, c)$
$\quad\quad \widehat{G}$ += $\{k \triangleright j \mid \overline{j} \in \widehat{F}(\overline{\mathsf{z}}) \wedge j \in \overline{j}\}$
$\quad\quad \widehat{G}$ += $\{i \triangleright j \mid i \in \widehat{F}(\mathsf{this_m}) \wedge j \in \widehat{F}(\mathsf{ret_n})\}$

(a) Object graph $\widehat{G}$.

(ACALL)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = dispatch(\mathsf{C}, c)$
$\quad\quad \widehat{P}_{k.\mathsf{n}}$ += $\{i.\mathsf{m} \mid i \in \widehat{F}(\mathsf{this_m})\}$

(ACALLTHIS)
c: $\mathsf{x} = \mathsf{this.n}(\overline{\mathsf{z}})$ in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = dispatch(\mathsf{C}, c)$
$\quad\quad \widehat{P}_{k.\mathsf{n}}$ += $\widehat{P}_{k.\mathsf{m}}$

(ACALL) and (ACALLTHIS)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$ in method $\mathsf{m} \Rightarrow$
$\quad$ *foreach* $k \in \widehat{F}(\mathsf{y})$
$\quad\quad \widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = dispatch(\mathsf{C}, c)$
$\quad\quad \widehat{C}$ += $\{i.\mathsf{m} \triangleright k.\mathsf{n} \mid i.\mathsf{m} \in \widehat{P}_{k.\mathsf{n}}\}$

(b) Call graph $\widehat{C}$.

Fig. 4. $\widehat{G}$, $\widehat{C}$, and each $\widehat{P}_{k.\mathsf{n}}$ are initialized to $\emptyset$. The rules (i.e., transfer functions) are applied iteratively until they reach fixpoint.

$i.\mathsf{m}$, where i is an abstract receiver of m, to $\widehat{P}_{k.\mathsf{n}}$. Rule (ACALLTHIS) adds $\widehat{P}_{k.\mathsf{m}}$, the set of caller tuples for $k.\mathsf{m}$ to $\widehat{P}_{k.\mathsf{n}}$. The last rule, applied to both (ACALL) and (ACALLTHIS), adds edges to $\widehat{C}$ from each tuple $i.\mathsf{m}$ in $\widehat{P}_{k.\mathsf{n}}$ to $k.\mathsf{n}$. Edges $i.\mathsf{m} \triangleright k.\mathsf{n}$ reflect that m, called on receiver i, calls n on receiver k (the edges "bypass" chains of calls on k through this). As it is customary with abstract interpretations, the rules (i.e., transfer functions) are applied repeatedly until $\widehat{P}_{k.\mathsf{n}}$ and $\widehat{C}$ reach fixpoint.

Note the explicit distinction of (ACALL) and (ACALLTHIS). A naive analysis will treat them identicaly, i.e., $\mathsf{y.n}()$ in method m would lead to edges from every tuple $i.\mathsf{m}$ where $i \in \widehat{F}(\mathsf{this_m})$ to every tuple $j.\mathsf{n}$, where $j \in \widehat{F}(\mathsf{y})$ regardless of whether y is this or not. $\widehat{F}(\mathsf{this_m})$ typically refers to a *set* of abstract objects. For example, if $\widehat{F}(\mathsf{this_m})$ is $\{i, j\}$ and the call is $\mathsf{this.n}()$, the naive analysis would lead to edges $i.\mathsf{m} \triangleright i.\mathsf{n}$, $i.\mathsf{m} \triangleright j.\mathsf{n}$, $j.\mathsf{m} \triangleright i.\mathsf{n}$ and $j.\mathsf{m} \triangleright j.\mathsf{n}$, when clearly, only the first and the last are feasible. A less naive analysis may make the distinction between (ACALL) and (ACALLTHIS), and at (ACALLTHIS) only create edges $i.\mathsf{m} \triangleright i.\mathsf{n}$, where $i \in \widehat{F}(\mathsf{this_m})$. This is not sufficient, because abstract edge $i.\mathsf{m} \triangleright i.\mathsf{n}$ may represent a call through this on the same concrete object, or a call from one concrete object i' to a different concrete object i'' where both i' and i'' are mapped to the same abstract i. Our analysis must capture transfer of control between distinct objects (i.e., inter-object transfer of control); by propagating the tuple that starts the chain of calls through this, it captures all inter-object transfer of control.

3 Dominance Inference Analysis

This section outlines dominance inference analysis and states its correctness results. This analysis is at the heart of object race detection (Section 4), but its details are beyond the scope of this paper. More details are available in [7].

We begin the description with several definitions. Let G be any directed graph. A *path* is written as $p = n_0 \triangleright n_1 \triangleright n_2 \triangleright \ldots n_{m-1} \triangleright n_m$; the trivial path is written as n_0 and a self-loop is written as $n_0 \triangleright n_0$. A root for G is a node $r \in G$ such that for all nodes $n \in G$ there is a (possibly trivial) path from r to n. A *boundary* for a node $n \in G$ is any graph $B_n \subseteq G$ such that n is a root of B_n. We assume that G has root root. A node $n \in G$ *dominates* node $n' \in G$ if all paths from root to n' go through n. The *dominance boundary* for a node $n \in G$ is the maximal boundary B_n such that for all nodes $n' \in B_n$, n *dominates* n' in G. We denote the dominance boundary of $n \in G$ as D_n.

3.1 Dominance Boundary

Dominance boundary analysis takes as input the abstract object graph $\widehat{G}$ and abstract object i, and computes $\widehat{B}_i \subseteq \widehat{G}$, the *abstract dominance boundary* of i. The following theorem holds for $\widehat{B}_i$:

Theorem 1. *Let G be any object graph and i be any object in G. Let B'_i be any boundary of i in G. If $\alpha(B'_i) \subseteq \widehat{B}_{\alpha(i)}$ then $B'_i \subseteq D_i$.*

The theorem states that the computed $\widehat{B}_{\alpha(i)}$ safely approximates the dominance boundary of i. That is, for any concrete boundary B'_i whose abstract representative is included in $\widehat{B}_{\alpha(i)}$, B'_i is included in D_i, or in other words, i dominates in G all of B'_i's nodes. Consider our running example. The abstract dominance boundary of object m, $\widehat{B}_m$, includes edges $m \triangleright b$, $m \triangleright t$ and $b \triangleright t$. The theorem states that every concrete m dominates the b and t objects it refers to.

3.2 Minimal Boundaries

Minimal boundary analysis takes as input an abstract object graph $\widehat{G}$ and an edge $i \triangleright j \in \widehat{G}$, and returns a set of objects, which we denote by $\widehat{minB}_{i \triangleright j}$. Each node $k \in \widehat{minB}_{i \triangleright j}$ is a root of a dominance boundary $\widehat{B}_k$ containing $i \triangleright j$. In our running example $\widehat{minB}_{b \triangleright t}$ equals $\{m\}$. Edge $b \triangleright t$ is contained in the boundary of root as well; however, the boundary of m is the minimal boundary. As another example, $\widehat{minB}_{t \triangleright w}$ equals $\{\mathsf{root}\}$.

Let G be any concrete object graph. We say that $k \in G$ *covers* $j \in G$ if for every path p from k to j, $p = k \triangleright \cdots \triangleright j$, $\alpha(p) \in \widehat{B}_{\alpha(k)}$. The following theorem ensures the safety of $\widehat{minB}$:

Theorem 2. *Let G be any concrete object graph and let $i \triangleright j \in G$ be any edge. There exists $k \in G$, $k \neq j$, such that (1) $\alpha(k) \in \widehat{minB}_{\alpha(i \triangleright j)}$ and (2) k covers j.*

The theorem guarantees the safety of the minimal boundary analysis. It states that $\widehat{minB}_{i \triangleright j}$ "covers" every concrete edge represented by $i \triangleright j$. In other words, for every concrete edge, we consider at least one root (and its boundary) that abstracts a *dominator* of that concrete edge (although not necessarily the immediate dominator). The theorem below ensures the minimality of $\widehat{minB}$:

Theorem 3. *Let $i \triangleright j \in G$ be any edge. Let $k \in G$, $k \neq j$ be such that (1) $\alpha(k) \in \widehat{minB}_{\alpha(i \triangleright j)}$ and (2) k covers j. For every k' if k dominates k' and k' covers j, then $\alpha(k') \in \widehat{minB}_{\alpha(i \triangleright j)}$.*

Informally, the theorem states that if there is a dominator k' which is closer than k, and k' covers j, then $\alpha(k')$ will be contained in $\widehat{minB}$.

4 Object Race Detection

We begin with the definition of an object race. In the style of [14], the execution of the program is viewed as a trace Tr of events $Tr = e_1, e_2 \cdots e_n$ performed by different threads. As in [14], an event is a tuple $e = (H, \overline{T}, l, t)$ which consists of a partial configuration $H; \overline{T}$, an action label l and a thread id t. An object race occurs when an event with a method call $\rightarrow j.\mathsf{n}$ occurs, and there is an outstanding call $j.\mathsf{n}'$ on the same receiver j made by a different thread (essentially, this is the complement of atomic set serializability as defined in [14]).

Definition 1. *There is an object race, denoted by $\langle j, \mathsf{n}, \mathsf{n}' \rangle$, when trace Tr contains event $e = (H, \overline{T}, j.\mathsf{n}, t)$, such that $\exists t' S \in \overline{T}$ where $t' \neq t$ and $\langle \mathsf{n}' \ F \ \mathsf{s} \rangle \in S$ and $F(\mathsf{this}) = j$.*

For convenience, we extend the above notation for events with calls to include the caller tuple. Namely, let event e correspond to step $H; \overline{T}; G; C; P \xrightarrow{\rightarrow j.\mathsf{n}}_t H'; \overline{T'}; G'; C'; P'$. Instead of $e = (H, \overline{T}, \rightarrow j.\mathsf{n}, t)$ we write $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ where $i.\mathsf{m} = P'$ (i.e., $i.\mathsf{m}$ started the chain of calls through this on receiver j).

An object race $\langle j, \mathsf{n}, \mathsf{n}' \rangle$ entails that there are paths $p = t.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n} \in C$ and $p' = t'.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n}' \in C$, where $t' \neq t$. Our object race detection analysis (Section 4.3) traverses pairs of abstract paths $p{=}t.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n} \in \widehat{C}$ and $p'{=}t'.\mathsf{run} \triangleright \cdots \triangleright j.\mathsf{n}' \in \widehat{C}$, where abstract t and t' are not necessarily different, and discovers object races. The analysis uses reentrancy analysis (Section 4.1), and lock analysis (Section 4.2) to avoid infeasible races. Non-reentrancy of edge $i \triangleright j$ guarantees (informally) that no two threads can execute events on $i \triangleright j$; thus, $i.\mathsf{m} \triangleright j.\mathsf{n} \in p$ and $i.\mathsf{m}' \triangleright j.\mathsf{n}' \in p'$ does not contribute an object race on j. Lock analysis associates locksets with events; non-empty intersection of two locksets guarantees (again informally) that events are executed serially.

4.1 Reentrancy Analysis

Reentrancy analysis computes predicate *reentrant*: $i \triangleright j \in \widehat{G} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ with the following properties. Let $i \triangleright j$ be any concrete edge, in any G. If *reentrant*($\alpha(i \triangleright$

j)) equals **false**, then (a) i creates j due to (DNEW) and (b) trace Tr does not contain a pair of events $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ and $e' = (H', \overline{T'}, i.\mathsf{m}' \triangleright j.\mathsf{n}', t')$ such that $t' \neq t$. Informally, if an edge is not reentrant, no two distinct threads can execute events on it.

We compute *reentrant* by first computing two sets of edges, *Fields* and *Flows*. Set *Fields* contains all abstract field edges:

$$\widehat{H}(i) = \mathsf{C}(...I_{\mathsf{f}}...) \ \wedge \ j \in I_{\mathsf{f}} \quad \Rightarrow \quad \mathit{Fields} \mathrel{+}= i \triangleright j$$

Set *Flows* contains all edges that capture object flow (i.e., object transfer from one object to another). Set *Flows* is computed during the construction of $\widehat{G}$; specifically, rule (ACALL) in Fig. 4(a) (left column) is augmented with the following two lines after the last line $\widehat{G}$ += ...:

$$\mathit{Flows} \mathrel{+}= \{k \triangleright j \mid \overline{j} \in \widehat{F}(\overline{\mathsf{z}}) \wedge j \in \overline{j}\}$$
$$\mathit{Flows} \mathrel{+}= \{i \triangleright j \mid i \in \widehat{F}(\mathsf{this}_{\mathsf{m}}) \wedge j \in \widehat{F}(\mathsf{ret}_{\mathsf{n}})\}$$

Objects j passed as arguments or returned at calls $\mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, are transferred, and the resulting edges are added to *Flows*. We now define *reentrant*:

$$\mathit{reentrant}(i \triangleright j) = \begin{cases} \textbf{true} & \text{if } i \triangleright j \in \mathit{Fields} \vee i \triangleright j \in \mathit{Flows} \\ \textbf{false} & \text{otherwise} \end{cases}$$

In our running example, edge $j \triangleright m$ is not reenetrant. In every call to run, thread j creates a new m object; m is not stored as a field of j and m does not flow back to j. It is impossible for one thread j to access another thread's m. Note that the m objects are not thread-local, because they escape to a field of static object c.

One can show that $\mathit{reentrant}(\alpha(i \triangleright j)) = \textbf{false}$ implies properties (a) and (b) stated at the beginning of this section. To show (b), suppose that there exists an edge $i \triangleright j$ in some G, such that $\mathit{reentrant}(\alpha(i \triangleright j)) = \textbf{false}$, and the trace contains events on $i \triangleright j$ $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ and $e' = (H', \overline{T'}, i.\mathsf{m}' \triangleright j.\mathsf{n}', t')$ such that $t' \neq t$. Let thread t create j, and consider thread t' and method m'. Roughly, m' can obtain a reference to j through object creation (rule (DNEW)), flow (rules (DCALL) and (DRET)) or field read ($\mathsf{x} = \mathsf{this.f}$); object creation is impossible because t created j, flow is impossible as well because then we would have had $\alpha(i \triangleright j) \in \mathit{Flows}$ and therefore $\mathit{reentrant}(\alpha(i \triangleright j))$ would have been **true**, and field read is impossible, because then we would have had $\alpha(i \triangleright j) \in \mathit{Fields}$ and again, $\mathit{reentrant}(\alpha(i \triangleright j))$ would have been **true**.

4.2 Lock Analysis

The lock analysis computes a map $\widehat{L}$ from abstract call graph edges to locksets. $\widehat{L}$ has the following property. Consider a pair of events with calls on object j: $e = (H, \overline{T}, i.\mathsf{m} \triangleright j.\mathsf{n}, t)$ and $e' = (H', \overline{T'}, i'.\mathsf{m}' \triangleright j.\mathsf{n}', t')$ such that $t' \neq t$. $\widehat{L}(\alpha(i.\mathsf{m} \triangleright j.\mathsf{n})) \cap \widehat{L}(\alpha(i'.\mathsf{m}' \triangleright j.\mathsf{n}')) \neq \emptyset$ implies that $j.\mathsf{n}$ and $j.\mathsf{n}'$ must be executed serially, or in other words, that events e and e' do not contribute an object race $\langle j, \mathsf{n}, \mathsf{n}' \rangle$. Informally, a non-empty intersection of two locksets guarantees serializability of the represented concrete events.

Source and Target Locksets. The first part of the lock analysis associates two sets, Sl_c (*source lockset*), and Tl_c (*target lockset*) to every call site c. Source lockset refers to the source tuple $i.\mathsf{m}$, and target lockset refers to the target tuple $k.\mathsf{n}$ in the call graph edge due to c. For example, call c in $\mathsf{synch\ (this)}\ \{ ...c\colon \mathsf{y.n}(\overline{\mathsf{z}})...\}$ in method m, leads to this being in Sl_c. this refers to the receiver of m; thus, for every edge $i.\mathsf{m} \triangleright k.\mathsf{n}$ due to c, the executing thread holds the lock on i before descending into the execution of $k.\mathsf{n}$.

All sets Sl_c and Tl_c are initialized to $\emptyset$ and updated as follows. If c occurs in a method m declared $\mathsf{synchronized}$, then this is added to Sl_c. If the target at c is declared $\mathsf{synchronized}$, then this is added to Tl_c. Additionally, we consider four patterns of usage of synchronized blocks. We note, however, that even without considering synchronized blocks, the lock analysis will be reasonably powerful, because synchronization in Java is naturally object-based (i.e., it is achieved by declaring methods as $\mathsf{synchronized}$).

Self locking occurs when the lock variable is this. We add this to Sl_c when the receiver variable $\mathsf{y} \neq \mathsf{this}$; we add this to Tl_c when the receiver variable is this.

$$\begin{aligned} \mathsf{synch\ (this)}\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \wedge \mathsf{y} \neq \mathsf{this} \quad &\Rightarrow \quad Sl_c \mathrel{+}= \mathsf{this} \\ \mathsf{synch\ (this)}\ \{...c\colon \mathsf{x} = \mathsf{this.n}(\overline{\mathsf{z}})...\} \quad &\Rightarrow \quad Tl_c \mathrel{+}= \mathsf{this} \end{aligned}$$

Global locking occurs when the lock variable lock is a static field, which is initialized exactly once during class initialization: $\mathsf{static\ C\ lock} = \mathsf{new\ C()};\ \boxed{l}$.

$$\mathsf{synch\ (lock)}\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \quad \Rightarrow \quad Tl_c \mathrel{+}= l$$

Local-object locking occurs when the lock variable lock is an instance field, which is initialized exactly once during object initialization: there is a field $\mathsf{C\ lock}$; and $\mathsf{lock} = \mathsf{new\ C()};\ \boxed{l}$ occurs in the constructor. We have:

$$\mathsf{synch\ (lock)}\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \wedge \mathsf{y} \neq \mathsf{this} \quad \Rightarrow \quad Sl_c \mathrel{+}= l$$

Client-side locking occurs when the lock variable y is the receiver at the call.

$$\mathsf{synch\ (y)}\ \{...c\colon \mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})...\} \quad \Rightarrow \quad Tl_c \mathrel{+}= \mathsf{this}$$

When y is a local variable, the addition of this is safe because stack locations in Java do not have aliases. When y is an instance field, however, the addition of this is not necessarily safe. In bytecode $\mathsf{synch\ (f)}\ \{...\mathsf{x} = \mathsf{f.n}(\overline{\mathsf{z}})...\}$ becomes $\mathsf{z} = \mathsf{this.f}; \mathsf{synch\ (z)}\ \{...\mathsf{w} = \mathsf{this.f}; \mathsf{w.n}(...)...\}$ and there could have been a write to f between the two reads. We add this to Tl_c only when our analysis proves that it is safe (e.g., f is initialized once in the constructor and is readonly afterwards).

Fig. 5 illustrates the above patterns of usage of synchronized blocks.

Map $\widehat{L}$. The second part of the lock analysis computes $\widehat{L}$, a map from call graph edges to locksets. The elements of a lockset are abstract objects plus this. For example, suppose that $\widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}) = \{l, \mathsf{this}\}$ where l is a global lock. The analysis guarantees that for every concrete edge $i'.\mathsf{m} \triangleright j'.\mathsf{n}$ represented by

```
class Account
  AccountImpl acc = ...;
  void update(int amt) {
    synch (this) {
   c:  x = acc.get();
       acc.put(amt+x);
    }
  }
```

(a) Self locking:
$Sl_c = \{\mathsf{this}\}$, $Tl_c = \emptyset$

```
class Account
  AccountImpl acc = ...
  Object lock = ... [l]
  void update(int amt) {
    synch (lock) {
   c:  x = acc.get();
       acc.put(amt+x);
    }
  }
```

(b) Local-object locking:
$Sl_c = \{l\}$, $Tl_c = \emptyset$

```
class Account
  AccountImpl acc = ...
  void update(int amt) {
    synch (acc) {
   c:  x = acc.get();
       acc.put(amt+x);
    }
  }
```

(c) Client-side locking:
$Sl_c = \emptyset$, $Tl_c = \{\mathsf{this}\}$

Fig. 5. Patterns of usage of synchronized blocks

(ACALL)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$, $\mathsf{y} \neq \mathsf{this}$, in method m $\Rightarrow$
  foreach $k \in \widehat{F}(\mathsf{y})$
    $\widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
    $\widehat{P}_{k.\mathsf{n}}$ += $\{(i.\mathsf{m}, Sl_c, Tl_c) \mid i \in \widehat{F}(\mathsf{this}_\mathsf{m})\}$

(ACALLTHIS)
c: $\mathsf{x} = \mathsf{this.n}(\overline{\mathsf{z}})$ in method m $\Rightarrow$
  foreach $k \in \widehat{F}(\mathsf{y})$
    $\widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
    $\widehat{P}_{k.\mathsf{n}}$ += $\{(i.\mathsf{m}', Sl \cup Sl_c, Tl \cup Tl_c) \mid (i.\mathsf{m}', Sl, Tl) \in \widehat{P}_{k.\mathsf{m}}\}$

(ACALL) and (ACALLTHIS)
c: $\mathsf{x} = \mathsf{y.n}(\overline{\mathsf{z}})$ in method m $\Rightarrow$
  foreach $k \in \widehat{F}(\mathsf{y})$
    $\widehat{H}(k) = \mathsf{C}(...) \quad \mathsf{n} = \mathit{dispatch}(\mathsf{C}, c)$
    foreach $(i.\mathsf{m}, Sl, Tl) \in \widehat{P}_{k.n}$
      $lockset = Tl \cup Tl_c$
      if $i \triangleright k \notin Flow \vee i \triangleright k \in \widehat{B}_i$
        if $\mathsf{this} \in Sl \cup Sl_c$
          $lockset$ += i
        $lockset$ += $(Sl \cup Sl_c) \backslash \{\mathsf{this}\}$
      $\widehat{C}$ += $\{i.\mathsf{m} \triangleright k.\mathsf{n} \mid (i.\mathsf{m}, Sl, Tl) \in \widehat{P}_{k.\mathsf{n}}\}$
      $\widehat{L}' = \widehat{L}[i.\mathsf{m} \triangleright k.\mathsf{n} \mapsto S]$, where
        $S = \widehat{L}(i.\mathsf{m} \triangleright k.\mathsf{n}) \cap lockset$

Fig. 6. Lock analysis. $\widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n})$ are initialized to the maximal set of locks. The rules are applied iteratively until they reach fixpoint.

$i.\mathsf{m} \triangleright j.\mathsf{n}$, the thread executing the edge holds the lock of l and the lock of j' before descending into the execution of $j'.\mathsf{n}$.

The analysis (Fig. 6) extends $\widehat{P}_{k.\mathsf{n}}$ to hold caller tuples $i.\mathsf{m}$ and the source and target locksets associated with $i.\mathsf{m}$. (ACALL) records Sl_c and Tl_c with $i.\mathsf{m}$. (ACALLTHIS) propagates the locksets of caller tuples down the this-call chain. For example, consider a method m which contains synch (lock) {... c: y.n ()...}, where lock is a global lock that points to l, and, in turn, n contains a call c': this.n′(). The analysis creates call graph edges $i.\mathsf{m} \triangleright j.\mathsf{n}'$ where i is an abstract receiver of m, and j' is a receiver at call site c'. The lock analysis propagates $Sl_c = \emptyset$ and $Tl_c = \{l\}$ to c' with $i.\mathsf{m}$; clearly, the call to $j.\mathsf{n}'$ from $i.\mathsf{m}$ is protected by the global lock.

The last rule (right column in Fig. 6) associates locksets to call graph edges. When adding an edge to $\widehat{C}$, it also computes a lockset, *lockset*, for that edge. The most interesting aspect of this rule is that Sl is propagated to *lockset* only if $i \triangleright k$ is not in *Flows* or it is in the boundary of i. This is necessary to ensure safety

of the analysis. If $i \triangleright k$ is in *Flows* and $i \triangleright k$ is not in the boundary of i, abstract object i may refer to different concrete objects, say i' and i'', with concrete edges $i' \triangleright k$ and $i'' \triangleright k$ both represented by the same abstract edge; thus, at runtime, this would refer to i' along edge $i' \triangleright k$, and to i'' along edge $i'' \triangleright k$; if one thread executes events along the first edge, and another thread executes events along the second edge, k remains unprotected. Analogous reasoning applies when *Sl* contains a local-object lock l. For example, consider Fig. 5(b). Let update be called with abstract receivers i and j, and let k be the abstract AccountImpl object. Assuming $i \triangleright k \in \widehat{B}_i$, we have $\widehat{L}(i.\mathsf{update} \triangleright k.\mathsf{get}) = \{l\}$.

The lock analysis is safe but not complete. It exploits the fact that synchronization in Java is naturally object-based (i.e., through this), and focuses on the "specialness" this. The special handling of this is what sets our analysis apart from other lock analyses such as [8] and [10] which aim at generality and appear to treat this as a regular reference variable. Our analysis is able to handle the vast majority of cases handled by the more complex analyses such as [8]. The empirical results confirm this conjecture. Furthermore, our analysis is extensible, as one can easily add new patterns of synchronized blocks.

4.3 Object Race Detection Analysis

The object race detection analysis is shown in Fig. 7. For ease of presentation, we make the following simplifying assumptions. First, there is a single static thread-fork site y.start, and it is located in main, second, the points-to set of y contains a single abstract object t, and three, run contains no calls through this. The analysis can be extended to handle arbitrarily many and arbitrarily located thread-fork sites, arbitrary points-to sets of y, and run methods that contain calls through this. Our implementation handles all cases.

The analysis computes a set of abstract races R. The following holds for R.

Theorem 4. *For every object race* $\langle j, \mathsf{n}, \mathsf{n}' \rangle$ *in trace Tr,* $\langle \alpha(j), \mathsf{n}, \mathsf{n}' \rangle \in R$.

Procedure *AllRaces* is the main driver. First, it identifies races on distributed objects (line 1); second, it descends into the boundary of each i to identify races on owned objects (lines 3-6). Procedure *DistributedRaces* maintains sets S_t and $S_{t'}$, which represent the accesses made by an arbitrary pair of threads t and t'. Note that each access is recorded with its lockset. Lines 3-12 traverse $\widehat{C}$ starting at t.run (i.e, the single thread-fork site y.start). The interesting part of this traversal is that j.n is added to $S_{t'}$ only if $i \triangleright j$ is reentrant; if $i \triangleright j$ is not reentrant, then threads cannot race on j along this edge (recall Section 4.1). In addition, *DistributedRaces* traverses $\widehat{C}$ starting at root.main and discovers accesses due to the main thread (lines 13-20). Finally, it computes and returns the set of races on distributed objects (line 21-22).

Procedure *Reach* takes as input a tuple $k.\mathsf{m}'$ and records all accesses $j.\mathsf{n}$ reachable from $k.\mathsf{m}'$ within the dominance boundary of k, $\widehat{B}_k$. $j.\mathsf{n}$ is recorded in S only if k is a minimal boundary of $i \triangleright j$ (recall minimal boundaries form Section 3.2). If k is not a minimal boundary, then there exists a smaller boundary, say of k'. The analysis will first discover races on the "closer dominator" k';

procedure *DistributedRaces*$(\widehat{G}, \widehat{C})$
output R
[1] $R = \emptyset, W = \{t.\mathsf{run}\}$
[2] $S_t = \emptyset, S_{t'} = \emptyset$
[3] while $W \neq \emptyset$
[4] remove $i.\mathsf{m}$ from W, mark $i.\mathsf{m}$ visited
[5] foreach $i.\mathsf{m} \triangleright j.\mathsf{n} \in \widehat{C}$
[6] if $\mathsf{root} \in \widehat{minB}_{i \triangleright j}$
[7] if *reentrant*$(i \triangleright j)$
[8] S_t += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[9] $S_{t'}$ += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[10] else
[11] S_t += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[12] if $j.\mathsf{n}$ not visited then W += $j.\mathsf{n}$

[13] $W = \{\mathsf{root.main}\}$
[14] while $W \neq \emptyset$
[15] remove $i.\mathsf{m}$ from W, mark $i.\mathsf{m}$ visited
[16] if $i.\mathsf{m} = t.\mathsf{run}$ continue
[17] foreach $i.\mathsf{m} \triangleright j.\mathsf{n} \in \widehat{C}$
[18] if $\mathsf{root} \in \widehat{minB}_{i \triangleright j}$
[19] $S_{t'}$ = += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[20] if $j.\mathsf{n}$ not visited then W += $j.\mathsf{n}$

[21] R += $\{\langle j, \mathsf{n}, \mathsf{n}' \rangle \mid (j.\mathsf{n}, ls) \in S_t \wedge (j.\mathsf{n}', ls') \in S_{t'} \wedge ls \cap ls' = \emptyset\}$
[22] return R

procedure *Reach*$(k.\mathsf{m}', \widehat{G}, \widehat{C})$
output S
[1] $S = \emptyset, W = \{k.\mathsf{m}'\}$
[2] while $W \neq \emptyset$
[3] remove $i.\mathsf{m}$ from W, mark $i.\mathsf{m}$ visited
[4] foreach $i.\mathsf{m} \triangleright j.\mathsf{m} \in \widehat{C}$ s.t. $i \triangleright j \in \widehat{B}_k$
[5] if $k \in \widehat{minB}_{i \triangleright j}$
[6] S += $(j.\mathsf{n}, \widehat{L}(i.\mathsf{m} \triangleright j.\mathsf{n}))$
[7] if $j.\mathsf{n}$ not visited then W += $j.\mathsf{n}$
[8] return S

procedure *AllRaces*$(\widehat{G}, \widehat{C})$
output R
[1] R = *DistributedRaces*$(\widehat{G}, \widehat{C})$
[2] while R changes
[3] foreach new race $\langle i, \mathsf{m}, \mathsf{m}' \rangle \in R$
[4] S_t = *Reach*$(i.\mathsf{m}, \widehat{G}, \widehat{C})$
[5] $S_{t'}$ = *Reach*$(i.\mathsf{m}', \widehat{G}, \widehat{C})$
[6] R += $\{\langle j, \mathsf{n}, \mathsf{n}' \rangle \mid (j.\mathsf{n}, ls) \in S_t \wedge (j.\mathsf{n}', ls') \in S_{t'} \wedge ls \cap ls' = \emptyset\}$
[7] return R

Fig. 7. Object race detection

eventual races with $j.\mathsf{n}$ will be discovered when the analysis descends into the boundary of k'.

5 Implementation

The object and call graph analyses, dominance analysis, and object race detection analysis are implemented in Java using Soot 2.2.3 [13] and Spark [5]. We performed whole-program analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a MacBook Pro laptop with a 2GHz Intel Core i7 processor and 4GB of RAM. The implementation, which includes Soot and Spark, was run with a max heap size of 1400MB; however, all benchmarks ran within a memory footprint of 800MB. Native methods are handled by utilizing the models provided by Soot. Reflection is handled by manually specifying the dynamically loaded classes. Our underlying points-to analysis analyzes constructors object-sensitively in the style of [6]. As a result, the running times reported for points-to analysis are approximately twice the running times of Spark.

Table 1. Results

Program	#Meth	ObjRace						Chord		Time[sec]	
		Distributed			Owned						
		Race-free	Racy		Race-free	Racy		False	Real	Points-to	Race
			False	Real		False	Real				
tsp	3414	5	**2**	0	2	0	0	**2**	0	35	1
hedc	3749	8	**2**	14	1	0	3	**2**	2	40	6
sor	3403	4	**2**	0	0	0	0	0	0	35	1
SPECjbb	4640	19	0	19	69	0	8	**30**	16	50	4
weblech	4461	3	0	8	3	0	0	0	2	52	3
jdbm	4331	1	0	4	25	0	0	0	2	45	3
jdbf	3994	90	0	20	0	0	2	**2**	4	55	5
commons	3551	0	**1**	8	13	0	0	0	7	42	2
jtds	5044	64	0	87?	10	0	63?	oom	oom	61	86

Our suite consists of benchmarks used in previous work on concurrency [15,9,14]. Column **#Meth** in Table 1 gives the size of the benchmarks in terms of the number of methods (user and library) reachable by Spark. Benchmarks tsp through weblech are whole-programs, and jdbm through jtds are libraries. For the libraries, we converted the single-threaded harnesses from [9] to the multithreaded model described in Section 4.3.

We compared our analysis with Chord [9], the leading static race detector. Chord's data race report includes a field-based view and an object-based view of data races. The object-based view groups data races per abstract object (distinguished by allocation site as in our analysis) and for each abstract object, the view provides a set of read/write access pairs. We counted each abstract object reported in the object-based view as a racy object. We used reports available at http://berkeley.intel-research.net/mnaik/research/pldi06_results.html for tsp and hedc. We ran Chord 2.0 and generated reports for the rest of the benchmarks, except for jtds, for which Chord ran out of memory with a max heap size 2GB. Both Chord and our analysis suppress race reports due to constructors and methods called within constructors.

Column **ObjRace** in Table 1 shows the number of non-thread-local (according to the escape analysis from [11]) objects, reported as race-free or racy by our analysis. Column **Chord** shows the number of racy objects reported by Chord. Our analysis classifies objects as Distributed or Owned. An object j is classified as Distributed when the test at line 6 in *DistributedRaces* fires **true**; it is classified as Owned otherwise. Note that, in general, an abstract object may be classified as both Distributed and Owned. When race-free, a distributed object is typically protected by its own lock (e.g., all methods called on that object are declared synchronized). In general, although there are race-free distributed objects, distributed objects tend to be racy. For owned objects (column Owned), when race free, an owned object is most often protected by synchronization on its owner. We observed many cases when synchronized methods access internal owned objects and the owned objects stay protected by their owner's lock. Although there are racy owned objects, owned objects tend to be race-free.

The authors examined the objects reported as racy by both our analysis and Chord, and classified those object races as false-positive (columns False), and feasible (columns Real). All feasible races reported by Chord were reported by our analysis as well. Our analysis reports more feasible races than Chord. One reason why feasible object races are not reported by Chord, is that although there is an object race, there is no immediate data race on the object's instance fields. In the majority of cases, an object race leads to calls that change state on internal unsynchronized library objects. The object races are symptoms not only of potential data races deeper in the boundary of the object, but of higher-level concurrency bugs such as atomicity errors and atomic serializability errors. We believe that it is valuable to report all object races. Another reason why feasible object races are not reported by Chord may be that Chord's lock analysis is unsafe [9], while our analysis is safe. There is a large number of racy objects reported on jtds, and we were unable to confirm with certainty whether those races were false or feasible; however, jtds is undersynchronized and it appears that the majority of the reported races are feasible.

6 Related Work

Concurrency is a large and active area of research and we cannot include a complete listing of related works. Below, we focus on the work closest to ours.

Von Pruan and Gross introduce the concept of the object race [15]. Their object race detection is dynamic. In fact, a primary goal is to optimize dynamic lockset- based race detection [12]; the higher-level concept of object race entails fewer dynamic checks and therefore lower overhead. In later work, von Praun and Gross introduce the concept of the Object Use Graph (OUG) [16] which allows reasoning about the temporal relation of object accesses, and further reduces the amount of dynamic checks in the lockset-based detector. Despite its name, the OUG is unrelated to the object graph from ownership types [3] that we infer. Our analysis reasons about object races as well. However, our analysis is entirely static. Furthermore, although [15] and [16] make use of "ownership", their notion of ownership is very different from ours. They refer to thread ownership, not dominance-based object ownership as we do. Similarly to [16], Choi et al. use static analysis as well as dynamic happens-before analysis, to optimize a dynamic lockset-based data race detector [2].

Chord [9] is the most advanced static race detector. Our work is similar in its goal: we wanted to build an effective static object race detector for Java. It is different from Chord in several ways. First, it focuses on object races while Chord focuses on data races. Second, our analysis uses a different algorithmic approach: it relies on *dominance analysis* at its heart, while Chord relies on *context-sensitive points-to analysis*. Dominance analysis entails a cheaper object abstraction — objects are represented by allocation site — which may lead to better scalability of our analysis. Our experiments indicate that our analysis is effective and that it complements Chord.

Work by Vaziri et al. [14] is most closely related to ours. It explores a type system for data-centric synchronization, and as in our work, dominance-based

ownership plays an important role. An object is viewed as an atomic set of fields, and the lock of that object protects its fields as well as internal (owned) objects. Work by Boyapati et al. [1] explores dominance-based ownership for safe multithreaded programming as well. In [14] and [1] ownership is specified by the programmer and checked by the type system. In contrast, we infer ownership and object races automatically.

7 Conclusions

We have presented a novel static object race detection analysis. We have shown its effectiveness by implementing a prototype, applying it to several large multithreaded Java benchmarks and comparing its results to the results of the leading static race detector Chord.

References

1. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 211–230 (2002)
2. Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: Conference on Programming Language Design and Implementation, pp. 252–269 (2002)
3. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 48–64 (1998)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In: Symposium on Principles of Programming Languages, pp. 238–252 (1977)
5. 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)
6. Milanova, A.: Light context-sensitive points-to analysis for java. In: Workshop on Program Analysis for Software Tools and Engineering, pp. 25–30 (2007)
7. Milanova, A., Vitek, J.: Static Dominance Inference. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 211–227. Springer, Heidelberg (2011)
8. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Symposium on Principles of Programming Languages, pp. 327–338 (2007)
9. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: Conference on Programming Language Design and Implementation, pp. 308–319 (2006)
10. Pratikakis, P., Foster, J., Hicks, M.: LOCKSMITH: context-sensitive correlation analysis for race detection. In: Conference on Programming Language Design and Implementation, pp. 320–331 (2006)
11. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java using annotated constraints. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 43–55 (October 2001)
12. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. In: Symposium on Operating Systems Principles, pp. 27–37 (1997)

13. Vallée-Rai, R., Gagnon, E.M., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
14. Vaziri, M., Tip, F., Dolby, J., Hammer, C., Vitek, J.: A Type System for Data-Centric Synchronization. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 304–328. Springer, Heidelberg (2010)
15. von Praun, C., Gross, T.: Object race detection. In: Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 70–82 (2001)
16. von Praun, C., Gross, T.: Static conflict analysis for multithreaded object-oriented programs. In: Conference on Programming Language Design and Implementation, pp. 115–128 (2003)

Soundness of Data Flow Analyses for Weak Memory Models*

Jade Alglave, Daniel Kroening, John Lugton,
Vincent Nimal, and Michael Tautschnig

Department of Computer Science, University of Oxford, UK

Abstract. Modern multi-core microprocessors implement *weak memory consistency models*; programming for these architectures is a challenge. This paper solves a problem open for ten years, and originally posed by Rinard: we identify sufficient conditions for a data flow analysis to be sound w.r.t. weak memory models. We first identify a class of analyses that are sound, and provide a formal proof of soundness at the level of trace semantics. Then we discuss how analyses unsound with respect to weak memory models can be repaired *via* a fixed point iteration, and provide experimental data on the runtime overhead of this method.

1 Introduction

Modern computing systems frequently employ multiple CPU cores, generating strong demand for concurrent software that exploits multiple threads of execution for better performance. However, the concurrency model implemented by these architectures is a formidable challenge for the programmer: with a goal of improving throughput, modern multi-core or multiprocessor architectures such as Intel's x86 series or IBM's PowerPC relinquish the standard execution model known as *Sequential Consistency* (SC) [1], in favour of much weaker models [2,3]. Multiprocessors featuring a *weak memory model* permit execution traces that do not correspond to any interleaving of the program's instructions, that is, the architecture does not implement SC.

Program bugs that relate to weak memory consistency are often difficult to reproduce and to diagnose. Fig. 1 shows a standard example to illustrate the problem. At line (a), processor P_0 writes the value 1 into memory address `x`; then at line (b) it reads from memory address `y` and writes the result into the processor-local register `r1`. Similarly, at line (c), processor P_1 writes the value 1 into memory address `y`; then at line (d)

Init: `x=0; y=0;`

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

Observed? `r1=0; r2=0;`

Fig. 1. Litmus test illustrating store buffering

* Supported by EPSRC under grants no. EP/G026254/1 and EP/H017585/1, by the ARTEMIS CESAR project, and under the European Union's Seventh Framework Programme (FP7/2007–2013)/ERC grant agreement no. 280053.

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 272–288, 2011.

it reads from memory address `x` and writes the result into processor-local register `r2`. We underline the fact that registers are private to the processor holding them, *e.g.*, `r1` is private to P_0, whereas the memory addresses, *e.g.*, `x` and `y`, are shared. Assuming SC, at least one of the registers has to hold 1 after the execution of the four statements. However, when executing this program on a multi-core x86 or PowerPC machine, traces are observed in which *both registers hold* 0 in the final state [4]. This outcome can be caused by the store buffers implemented in these architectures. The situation is exacerbated by the fact that this non-SC observation only occurs in a small fraction of the executions. For instance, execution of the code of Fig. 1 using the `litmus` tool presented in [5] on an x86 system results in 99.13% SC-conforming traces among one billion executions.

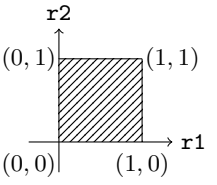
These relaxations permitted in weak memory models affect the semantics of high-level languages such as Java [6] or C++ [7]. One way to address this issue is to restrict program analyses to programs that are guaranteed to only exhibit SC executions such as programs free of data races [8], or programs where memory barriers have been inserted to ensure that they only have SC executions [9,10].

Yet we cannot restrict ourselves to this limited view on programs: engineers often choose to retain non-SC executions for performance reasons. *In other terms, we do* **not** *restrict our study to data-race free programs*. Consequently, effects relating to weak memory consistency need to be modelled appropriately in program analysis algorithms for concurrent software. The issue of soundness of program analyses w.r.t. weak memory models has been identified, among others, by Rinard, who wrote ten years ago in [11]:

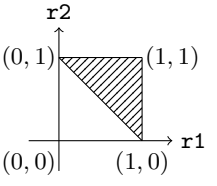
> *"We suspect that many existing analyses are sound for programs with weak consistency models [...], but this soundness is clearly inadvertent, in some cases a consequence of the imprecision in the analysis, and not necessarily obvious to prove formally."*

As an example, we first perform an *interval* analysis [12] on Fig. 1, to determine the possible values of `r1` and `r2`. We compute an interval for each variable. The *join* of two intervals yields the smallest interval that contains both of them. We consider all possible interleavings of statements of the two threads and compute the join over all these traces. For instance, for the traces $(a);(b);(c);(d)$ and $(c);(d);(a);(b)$ we obtain the intervals $[0,0]\times[1,1]$ and $[1,1]\times[0,0]$, respectively. The join $[1,1]\times[0,0]\sqcup[0,0]\times[1,1]$ yields the box $[0,1]\times[0,1]$, already including the result that can be derived from the other interleavings, *i.e.*, $[1,1]\times[1,1]$. More interestingly, this overapproximation also includes the additional value that one can observe on a weak memory model, *i.e.*, $(0,0)$.

As a second example of a program analysis, we consider the *octagon abstract domain* [13], which is a relational domain that describes (octagonal) faces of polyhedra. *Joining* two polyhedra in this abstraction consists in computing the smallest polyhedron which contains these polyhedra. For the two traces given above we obtain $\{1\}\times\{0\}\sqcup\{0\}\times\{1\} = \{\texttt{r1}+\texttt{r2}\leq 1, -\texttt{r1}-\texttt{r2}\leq -1, \texttt{r1}\leq 1, -\texttt{r1}\leq 0\}$, which concretizes to the diagonal line segment going from $(0,1)$ to $(1,0)$. *No join of interleavings, however, will include the point* $(0,0)$.



(a) Box abstraction



(b) Octagon abstraction

```
var x,y:int; // shared memory
initial x==0 and y==0;
thread P0:
var r1:int; // P0 register
begin
  x = 1;
  r1 = y;
end
thread P1:
var r2:int; // P1 register
begin
  y = 1;
  r2 = x;
end
```

(c) Implementation of Fig. 1 for ConcurInterproc

Fig. 2. Running interval and octagon on Fig. 1 to compute the values of (r1, r2)

Fig. 2 provides a comparison of the results of intervals and octagons. In Fig. 2(c) we furthermore provide the code for reproducing these results using ConcurInterproc[1] [14]. The octagon domain is thus unsound w.r.t. weak memory models, whereas the (less precise) interval domain belongs to a class of analyses sound for weak memory models, as we show in this paper.

Few proofs of soundness of program analyses for weak memory models exist. In addition, existing proofs are usually tailored to a particular analysis and a particular memory model [15,16]. These proofs thus offer only limited general insight into what makes an analysis sound for weak memory models.

Contributions. We establish sufficient conditions for a data flow analysis to be sound w.r.t. weak memory models. We identify a large class of data flow analyses—the *non-relational* ones—that satisfy these conditions. These are guaranteed to be sound for a wide range of modern architectures, namely all those that respect the uniproc axiom as defined in [17,4] and recalled in Sec. 2. Our results use trace semantics, hence are independent of the programming language and the specific representation of the concurrent program used in the analysis. Our classification confirms recent research results for specific analyses [15,16,18,19] as part of a broader result. It also simplifies existing ad-hoc proofs, and provides proofs that new analyses are sound w.r.t. weak memory models.

We furthermore address the question of repairing an analysis that is unsound for weak memory models. We provide a general method to extend a sequentially sound *forward* analysis to an analysis for concurrent programs that is sound for weak memory models. We illustrate the method with the octagon domain.

[1] http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi

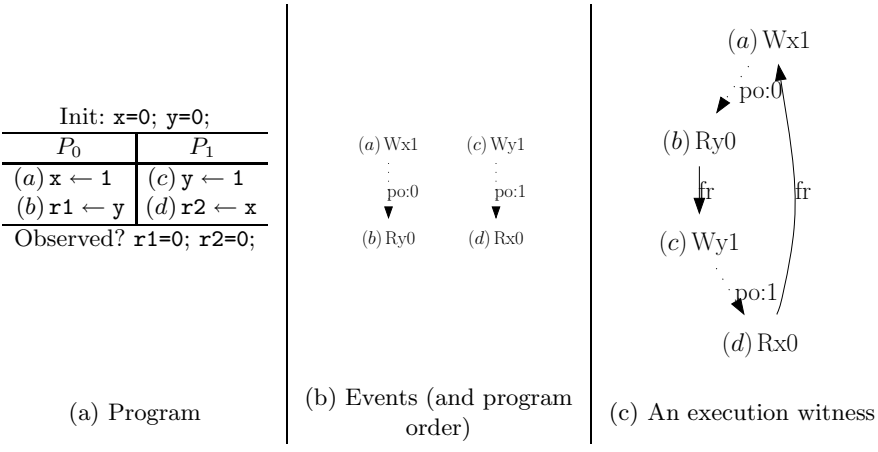


Fig. 3. A program and a candidate execution

We omit the proofs for brevity, but they can be found together with the details of our experiments at `http://www.cprover.org/wmm/`.

2 Background

To apply program analyses to concurrent programs running on modern multicore processors or multiprocessors, we need to prove that these analyses are actually sound w.r.t. weak memory models. To describe weak memory models, we use the generic framework of Alglave *et al.* [4,17], which covers a wide range of existing architectures, in particular x86-TSO [20] and a fragment of Power. We summarise the relevant parts of this framework.

2.1 Weak Memory Models

Events. Instead of dealing directly with programs, we reason in terms of the *events* occurring in a program execution. An event e is a memory access, composed of a direction R (read) or W (write), an address addr(e), a value val(e), a processor proc(e), and a program location loc(e). We will use registers, which are processor-local (thread-local) variables, in place of values when the actual valuation is not known *a priori*. Note that an address always refers to shared memory, and thus never to a register. We represent each instruction by the events it issues. In Fig. 3, we associate the store (a) `x` $\leftarrow$ `1` on P_0 with the event $e = (a)\mathrm{Wx1}$. For this example we have addr(e) = `x`, val(e) = 1, proc(e) = P_0, and loc(e) =(a).We write $\mathbb{E}$ for the set of events. We write w (resp. r) for a write (resp. read), and e when the direction of the event is irrelevant.

Executions. We associate a program with an *event structure* $E \triangleq (\mathbb{E}, \overset{\text{po}}{\rightarrow})$, composed of its events $\mathbb{E}$ and the *program order* $\overset{\text{po}}{\rightarrow}$, a per-processor total order over $\mathbb{E}$. In Fig. 3, the store (a) to `x` on P_0 is in program order with the read (b) from `y` on P_0, *i.e.*, (a)W`x1` $\overset{\text{po}}{\rightarrow}$ (b)R`y0`.

Given an event structure E, we represent an execution witness $X \triangleq (\overset{\text{ws}}{\rightarrow}, \overset{\text{rf}}{\rightarrow})$ of the corresponding program by two relations over $\mathbb{E}$: the *write serialisation* $\overset{\text{ws}}{\rightarrow}$ is a per-address total order on writes, linking a write w to all other writes w' to the same address hitting the memory after w; the *read-from map* $\overset{\text{rf}}{\rightarrow}$ links a single write w to a read event r that reads from the address that w writes to. The relations $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{rf}}{\rightarrow}$ are the key objects for defining the validity of an execution, as explained below. We derive the *from-read map* $\overset{\text{fr}}{\rightarrow}$ from $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{rf}}{\rightarrow}$. A read r is in $\overset{\text{fr}}{\rightarrow}$ with a write w when r reads from the address of some write w' that hits the memory before w does: $r \overset{\text{fr}}{\rightarrow} w \triangleq \exists w'.\, w' \overset{\text{rf}}{\rightarrow} r \wedge w' \overset{\text{ws}}{\rightarrow} w$.

The observable result, `r1=r2=0`, shown in Fig. 3(a) corresponds to the execution of Fig. 3(c) if each address and register initially holds 0. If `r1=0` in the end, the read (b) obtained its value from the initial state, hence before the write (c) on P_1, thus $(b) \overset{\text{fr}}{\rightarrow} (c)$. Similarly, if `r2=0`, then $(d) \overset{\text{fr}}{\rightarrow} (a)$.

Uniprocessor Behaviour. The condition uniproc$(E, X) \triangleq$ acyclic$(\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow} \cup \overset{\text{po-loc}}{\rightarrow})$ (where $\overset{\text{po-loc}}{\rightarrow}$ is the program order restricted to events with the same address) forces a processor in a multiprocessor context to respect the memory coherence widely assumed by modern architectures [21,22,2,3].

This means that if a processor writes, *e.g.*, the value v to the memory location ℓ and then reads v' from ℓ, then the associated writes w and w' should be in this order in the write serialisation, *i.e.*, w' should not precede w. In Fig. 4, we have $(c) \overset{\text{ws}}{\rightarrow} (a)$ (by `x` final value) and $(a) \overset{\text{rf}}{\rightarrow} (b)$ (by `r1` final value). The cycle $(a) \overset{\text{rf}}{\rightarrow} (b) \overset{\text{po-loc}}{\rightarrow} (c) \overset{\text{ws}}{\rightarrow} (a)$ invalidates this execution: (b) cannot read from (a) as it is a future value of `x` in $\overset{\text{ws}}{\rightarrow}$. In every model of our framework, there is no valid execution which ends up with $x = 1$, `r1` $= 1$.

P_0	P_1
(a) `x` $\leftarrow 1$	(b) `r1` $\leftarrow$ `x`
	(c) `x` $\leftarrow 2$
Forbidden: `x=1; r1=1;`	

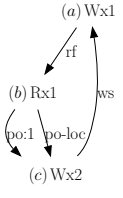


Fig. 4. An invalid execution, violating the uniproc condition.

The uniproc condition actually corresponds to checking that SC holds per address [17]. We rely heavily on this axiom in the proofs of this paper[2].

Architectures; Validity of Executions. We define formally in [4,17] the notions of *architecture* and *validity of an execution w.r.t. an architecture*, but we abstract them away in the present paper, for two reasons. First, the exposition of this

[2] All the results presented here hold with a weaker version of uniproc, that allows us to embrace Sun's RMO in our framework. We omit this restriction for clarity and brevity, but more details can be found in [17, p.47-48].

paper does not need to detail them. Second, and more importantly, our results only require the architecture that we consider to respect the uniproc axiom.

Thus, in the following, we consider an abstract notion of *architecture*, which acts as a filter over executions. Given an architecture A, an event structure E and an execution witness X, we write $\text{valid}_A(E, X)$ when the execution (E, X) is valid on A. We only impose that $\text{valid}_A(E, X)$ implies $\text{uniproc}(E, X)$, *i.e.*, that a valid execution should pass the uniproc check.

We also abstract the notion of comparison over architectures of [4,17]. Intuitively, an architecture A_1 is *weaker* than another one A_2 when the executions valid on A_2 are valid on A_1. Thus, SC is stronger than any other architecture.

2.2 Programs vs. Event Structures

Event structures describe programs in terms of their trace semantics. In the programs considered above, each right hand side of a store was a single concrete value, which immediately translated to a (concrete) event. To derive event structures from a description of general high-level programs, however, we proceed in two steps. Each control flow path at program level first translates to an *abstract event structure*, where events take the form of a direction and two variables. This allows us to translate, *e.g.*, a store $(a)\ x \leftarrow \sigma$ to the (abstract) event $(a)\mathrm{W}x\sigma$.

We write $\mathcal{E}$ for the set of all abstract event structures, $\mathbb{A}$ for the set of all addresses, and $\mathbb{V}$ for the set of all values. We define the type $\mathcal{R}$ of *results* (or valuations) as $\mathcal{R} \triangleq \wp(\mathbb{A} \times \mathbb{V})$, *i.e.*, a result is a set of pairs (x, v) where x is an address and v a value (we denote the powerset with $\wp(\cdot)$).

Given a specific language $\mathcal{L}$, we write $\mathbb{P}_\mathcal{L}$ for the set of all the programs which can be written in this language. We introduce $\alpha : \mathbb{P}_\mathcal{L} \to \wp(\mathcal{E} \times \mathcal{R})$, which maps a program $\mathcal{P}$ to corresponding abstract event structures and initial values, respecting the semantics of the language $\mathcal{L}$. Each event created by α is labelled by the program counter of the corresponding statement in $\mathcal{P}$.

Each abstract event structure induces multiple *concrete event structures* under a given set of initial valuations. That is, an abstract event $(a)\mathrm{W}x\sigma$ with $R = \{(\sigma, 0), (\sigma, 1)\}$ translates to concrete events $(a)\mathrm{W}x0$ and $(a)\mathrm{W}x1$. The set of all sets of concrete event structures is denoted by $\mathcal{E}_{\text{conc}}$. We use the mapping $\text{conc} : \mathcal{E} \to \mathcal{R} \to \mathcal{E}_{\text{conc}}$ to translate abstract to concrete event structures. We require conc to yield a set of concrete event structures such that at least for each execution witness valid on an architecture A there is a concrete event structure.

We distinguish abstract from concrete event structures as follows: program analyses will be applied to abstract event structures, but reasoning about actual values will be performed in concrete event structures.

3 Soundness of Analyses on Weak Memory Models

We define an *analysis* $[\![\cdot]\!]$ as mapping abstract event structures and initial valuations to sets of pairs (i, r) where i is a program location (of type $\mathbb{L}$) and r is a

result as defined in the preceding section. We make explicit the initial state of values of type $\mathcal{R}$, commonly being the empty set or the set of all possible values:

$$[\![\cdot]\!] : \mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times \mathcal{R})$$

Note that our definition captures *relational* analyses [23]; indeed the result type $\wp(\mathbb{L} \times \mathcal{R})$ can be rewritten as $\mathbb{L} \to \wp(\mathbb{A} \to \wp(\mathbb{V}))$.

Consider an abstract event $(a)\mathrm{W}x\sigma$ with initial valuations $R = \{(\sigma, 0), (\sigma, 1)\}$. We track the concrete values of x and the relation between x and σ as follows:

$$[\![(a)\mathrm{W}x\sigma, \{(\sigma, 0), (\sigma, 1)\}]\!] = \{((a), \{(x, 0), (\sigma, 0)\}), ((a), \{(x, 1), (\sigma, 1)\})\}$$

3.1 Definition of Soundness

Rinard and Rugina define in [24, A.3] an analysis to be *sound*

> *"[. . .] if it is at least as conservative as the result obtained by using the standard pointer analysis algorithm for sequential programs on all the interleavings of the legal executions."*

A *legal execution* corresponds to the execution of one thread. Thus their work assumes SC (*i.e.*, the interleaving semantics) as the execution model. We generalise their idea to weak memory models. Given an architecture A, we write $\mathrm{values}_A(E, R)$ for the set of values that execution witnesses X can yield on A, where X is an execution witness associated to a concrete event structure $E' \in \mathrm{conc}(E, R)$, *i.e.*, obtained from concretizing E with initial valuations R.

We write $<_X$ for the order on program locations induced by an execution X. We omit the formal definition of $<_X$ for brevity; it corresponds to $A.\mathrm{ghb}(E, X)$ as defined in [17,4]. Intuitively, it describes the order in which the memory events of X hit the memory. For example in Fig. 3(c), on an architecture $A \neq \mathrm{SC}$—for otherwise the execution X depicted would not be valid—$<_X$ corresponds to $\{((d), (a))\} \cup \{((b), (c))\}$.

We write $\mathrm{last}(r, i, x)$ when the location of x is less than (or unrelated to) (i) in $<_X$, and x is one of the last elements in the relation r, *i.e.*, there is no element x' such that $(x, x') \in r$. If a given write $w = (i)\mathrm{W}xv$ is the last element in $\mathrm{ws}(X)$ at location (i), then the value v is the *current value of* x *at location* (i). For example in Fig. 4, the current value of x at line (c) (resp. (a)) is 1 (resp. 2), for the last write to x at line (c) (resp. (a)) is the write $(c)\mathrm{W}x2$ (resp. $(a)\mathrm{W}x1$).

Thus, we define $\mathrm{values}_A(E, R)$ as the set of possible results, *i.e.*, mappings of each address to its current value, at each location. In other words, the set $\mathrm{values}_A(E, R)$ collects all the possible values in memory addresses that can arise in an execution (E, X) that is valid on A:

$$\begin{aligned}\mathrm{values}_A(E, R) \triangleq \{(i, r) \mid \exists X.\, \mathrm{valid}_A(\mathrm{conc}(E, R), X) \wedge \forall x, v.\, (x, v) \in r \Rightarrow \\ \exists w.\, (\mathrm{last}(\mathrm{ws}(X), i, w) \wedge \mathrm{addr}(w) = x \wedge \mathrm{val}(w) = v)\}\end{aligned}$$

For example in Fig. 3, $\mathrm{values}_{SC}(E, R)$ contains, for program location (a), the result $((a), \{(\mathtt{r1}, 0), (\mathtt{r2}, 0), (x, 1), (y, 0)\})$. Formally, we define soundness of over-approximating analyses for a weak architecture A as follows:

Definition 1. *An analysis* $[\![\cdot]\!]$ *is* A-sound *iff the result of* $[\![\cdot]\!]$ *on an abstract event structure* E *with initial values* R *describes a state space at least as large as that of* $\text{values}_A(E,R)$ *(with* $U \preceq V$ *iff* $\forall (i,r) \in U.\, \exists r'.(i,r') \in V \wedge r' \subseteq r$*):*

$$\text{sound}_A([\![\cdot]\!]) \triangleq \forall E, R.\, \text{values}_A(E,R) \preceq [\![E,R]\!]$$

We have, *e.g.*, $\{((a),\{(\texttt{r1},0),(\texttt{r2},0),(x,1),(y,0)\})\} \preceq \{((a),\{(\texttt{r1},0),(\texttt{r2},0)\})$. This means that we consider an analysis result to be A-sound if it is at least as conservative as taking all the values yielded by all valid executions on A.

Note that under-approximating analyses for SC are also under-approximating for all weak memory models, since for all weak architectures A, the values valid on SC are also valid on A, *i.e.*, $\text{values}_{\text{SC}}(E,R) \preceq \text{values}_A(E,R)$. We therefore focus the presentation on showing soundness of over-approximating analyses.

3.2 SC-Soundness Entails *A*-Soundness for Non-relational Analyses

We now define a particular class of program analyses by restricting the signature of the output of the analysis. We only consider analyses $\widehat{[\![\cdot]\!]}$ that map abstract event structures to pairs (i,r) where i is a program location and r a result, with the additional constraint that r *is a singleton*:

$$\widehat{[\![\cdot]\!]} : \mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times (\mathbb{A} \times \mathbb{V}))$$

This type can be rewritten as $\mathbb{L} \to (\mathbb{A} \to \wp(\mathbb{V}))$. In practice, we apply a *projection* to our general type of analyses to obtain *non-relational* ones [23]:

$$\text{projection}([\![\cdot]\!])(E,R) \triangleq \{(i,\{(x,v)\}) \mid \exists r.\, (x,v) \in r \wedge (i,r) \in [\![E,R]\!]\}$$

We restrict the type of values_A similarly by computing $\widehat{\text{values}_A}$ and then using the projection abstraction given above. We write $\widehat{\text{values}_A}(E,R)$ to indicate this, *i.e.*, $\widehat{\text{values}_A}(E,R)$ is of type $\mathbb{L} \to (\mathbb{A} \to \wp(\mathbb{V}))$. In the example of Fig. 3, $\widehat{\text{values}_{\text{SC}}}(E,R)$ contains $\{((a),(\texttt{r1},0)),((a),(\texttt{r2},0)),((a),(x,1)),((a),(y,0))\}$.

We want to determine when a given analysis, although designed with SC in mind, is sound for a weak architecture A. For example, for the program given in Fig. 3, we have $\widehat{\text{values}_{\text{x86}}}(E,R) = \widehat{\text{values}_{\text{SC}}}(E,R)$ (with the initial state R mapping all variables to 0). Hence in this case, an analysis that computes at least $\widehat{\text{values}_{\text{SC}}}(E,R)$ is also sound for x86, since it also computes all the values that this specific program can yield on an x86 machine. We show that *any analysis* $\widehat{[\![\cdot]\!]}$ with (1) matching signature and (2) that is SC-sound as defined above satisfies this requirement. This means that collecting the values produced by the SC executions (*i.e.*, $\widehat{\text{values}_{\text{SC}}}(E,R)$) suffices to obtain the values yielded by a weaker model A. This property is guaranteed by the uniproc check as defined in Sec. 2, since uniproc means that SC holds per location. To prove this claim, we first show the inclusion of value sets:

Lemma 1. $\forall E, R.\, \widehat{\text{values}}_A(E,R) \subseteq \widehat{\text{values}}_{SC}(E,R)$

The lemma is sufficient to show our main theorem, which states that for a non-relational analysis $\widehat{[\![\cdot]\!]}$, its SC-soundness (*i.e.*, $\forall E, R.\, \widehat{\text{values}}_{\text{SC}}(E,R) \subseteq \widehat{[\![E,R]\!]}$) entails its A-soundness on any architecture A. That is to say, we show in Thm. 1 that a non-relational analysis, though defined with SC in mind, is sound on a weaker architecture A when this analysis collects at least all the values yielded by all the executions valid on SC. We formalise this as follows.

Theorem 1. $\forall \widehat{[\![\cdot]\!]} : \mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times (\mathbb{A} \times \mathbb{V})).\ \text{sound}_{SC}(\widehat{[\![\cdot]\!]}) \Rightarrow \text{sound}_A(\widehat{[\![\cdot]\!]})$

The result is obtained by reasoning over traces, which is the most precise, yet not necessarily computable, representation for program executions. Hence our results are *independent of (1) programming language specifics* such as locks or dynamic synchronization primitives and hold for all other program representations, such as (concurrent) control flow graphs or Petri nets, for they are overapproximations of the sets of traces (*cf.* [25] for a discussion of representations for concurrent programs); *(2) analysis specifics* such as fixed point iteration strategies and sources of imprecision.

Note that for a relational analysis, its SC-soundness would not, in general, entail its A-soundness, for the weaknesses of multiprocessors' execution models is precisely observable *via* relations over variables, as shown in Fig. 3. We discuss means of obtaining an A-sound analysis from a relational analysis in Sec. 4.

Octagon and Box. As seen in Sec. 1, the octagon abstract interpretation is not sound on weak memory models. Indeed, this analysis reasons over conjunctions of statements, *e.g.*, for Fig. 3, it computes values that `r1` and `r2` can have at the same time. More formally, in [13], Miné defines the octagon concretization function with $\mathcal{D}^+ : DBM \to \wp(\mathbb{A} \to \mathbb{V})$, where DBM is the set of difference-bound matrices m^+. There is one matrix m_i^+ per line i, and the concrete domain computation takes the form $\lambda i.\mathcal{D}^+(m_i^+) : \mathbb{L} \to \wp(\mathbb{A} \to \mathbb{V})$. We have $\wp(\mathbb{A} \times \mathbb{V}) \subsetneq \wp(\mathbb{A} \to \mathbb{V}) \subsetneq \wp(\wp(\mathbb{A} \times \mathbb{V}))$, hence octagon analyses cannot be represented with the non-relational analysis type, $\mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times (\mathbb{A} \times \mathbb{V}))$, but always with the relational analysis type, *i.e.*, $\mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times \wp(\mathbb{A} \times \mathbb{V}))$.

As we show in the introduction, the interval abstraction, however, collects along the way the values that `r1` and `r2` can have on a weak memory model. This is a non-relational analysis, expressible with $\mathcal{E} \to \mathcal{R} \to \mathbb{L} \to \wp(\mathbb{A} \times \mathbb{V})$, and we can deduce from Thm. 1 that it is sound for weaker memory models if originally implemented for SC.

3.3 Proving Soundness of Analyses over Programs

Thm. 1 gives sufficient conditions for an analysis over event structures to be A-sound. We explain here how this result transfers to programs.

Let $\mathcal{P}$ be a program written in a language $\mathcal{L}$. To express the soundness of a program analysis $[\![\cdot]\!]_{\mathcal{L}}$, we require $\text{values}_A(E,R)$ and $[\![\mathcal{P}]\!]_{\mathcal{L}}$ to have the same type $\wp(\mathbb{L} \times \mathcal{R})$, with $\mathbb{L}$ being the program counters of statements in program $\mathcal{P}$.

As above, we define $\widetilde{\text{values}}_A(\mathcal{P})$ as the values yielded by executions of $\mathcal{P}$ on A, *i.e.*, $\widetilde{\text{values}}_A(\mathcal{P}) \triangleq \bigcup_{(E,R)\in\alpha(\mathcal{P})} \text{values}_A(E,R)$ (recall that $\alpha : \mathbb{P}_{\mathcal{L}} \to \wp(\mathcal{E}\times\mathcal{R})$ maps a program $\mathcal{P}$ to corresponding abstract event structures and initial values, w.r.t. the semantics of the language $\mathcal{L}$).

Hence the A-soundness of a program analysis is merely a lifting of the A-soundness of the corresponding event structure analysis:

$$\widetilde{\text{sound}}_A([\![\cdot]\!]_{\mathcal{L}}) \triangleq \forall\mathcal{P}.\, \widetilde{\text{values}}_A(\mathcal{P}) \preceq [\![\mathcal{P}]\!]_{\mathcal{L}}$$

Therefore, the A-soundness of SC-sound non-relational program analyses holds as a corollary of Thm. 1:

Corollary 1. $\forall [\![\cdot]\!]_{\mathcal{L}} : \mathbb{P}_{\mathcal{L}} \to \wp(\mathbb{L}\times(\mathbb{A}\times\mathbb{V})).\, \widetilde{\text{sound}}_{SC}([\![\cdot]\!]_{\mathcal{L}}) \Rightarrow \widetilde{\text{sound}}_A([\![\cdot]\!]_{\mathcal{L}})$

Rugina and Rinard's Pointer Analysis. Rugina and Rinard define in [24] a non-relational analysis (denoted RR in the following) for a subset of C with basic pointer assignments and control flow instructions. They prove that $RR(\mathcal{P})$ contains all the values appearing in the interleavings of the program $\mathcal{P}$, *i.e.*, RR is SC-sound. Thus by Cor. 1, RR is sound for memory models weaker than SC.

4 Repairing Unsound Analyses

We have shown that a non-relational analysis is A-sound if it is SC-sound. Yet some analyses, such as the octagon one, cannot be defined in the non-relational framework. Using the projection abstraction defined in Sec. 3.2, we may turn a relational analysis into a non-relational one. Thus, projecting an unsound analysis makes it sound (providing it is SC-sound) by Thm. 1.

Yet, this projection is very coarse, as it breaks all the relations over variables maintained by the analysis (*cf.* [13] for an example of the projection from octagon to interval). We present in the following a method to ensure the soundness of an analysis w.r.t. weak memory models, which preserves the relational type of the analysis, and conserves some of its precision. This method, which we call *repair loop*, is already implemented in several existing analyses, for performance reasons. Its consists of analysing each thread separately to capture the values the memory locations get, then feeding back the collected possible values to each of the other threads to simulate the effects of thread interference (*cf.* Sec. 6 for a discussion of several approaches already following this idea). We choose to simulate the process by:

1. building enough *concatenations of the threads* of a program;
2. analysing each of these as a sole thread *without killing any values*.

Concatenations of Threads. We now assume that our event structures are *finite*, *i.e.*, have an arbitrary large yet finite number of events. We say that an event structure is a *thread* when all of its events belong to the same processor. Given an event structure E, a thread corresponds to the restriction of E to the events

that run on processor p, written E_p. The *sequence* of two threads E_i and E_j is itself a thread, in the sense that it has only one processor; it gathers both the events of E_i and E_j, and its program order corresponds to the program order of E_i followed by the program order of E_j. We write e_i (resp. e_j) for the last (resp. first) event in program order on E_i (resp. E_j). We write evts(E) (resp. procs(E)) for the events (resp. processors) of a given event structure E:

$$E_i; E_j \triangleq \big(\text{evts}(E_i) \cup \text{evts}(E_j), \text{po}(E_i) \cup \text{po}(E_j) \cup \{(e_i, e_j)\}\big)$$

We further define the *concatenation* of n threads of an event structure E as:

$$\text{concat}(E, n) \triangleq \{E_{\text{big}} \mid \exists T_1, \ldots, T_n.\, E_{\text{big}} = T_1; \ldots; T_n \wedge \forall i.\, \exists p \in \text{procs}(E).\, T_i = E_p\}$$

We prove that for a finite E there exists an integer n_{SC} (bounded by the cardinality of evts(E)) such that *each interleaving (an SC-valid execution) can be found as a subsequence* of some E_{big} in concat(E, n_{SC}). For example in Fig. 3, we simulate the interleaving $(a), (b), (c), (d)$ by building the concatenation $P_0; P_0; P_1; P_1$. Note that, since the E_{big} are themselves threads, we can analyse them with a sequential analysis. Thus, analysing all the E_{big} of concat(E, n_{SC}) gives us all the possible values yielded by the interleavings of E.

Analyse without Killing. We define here what it means to analyse one thread without killing any values. We give in Alg. 1 the code of the recursive function awk. The function applies the analysis on the thread and propagates its relations to collect all the results.

$$\begin{array}{l}\text{awk } (\llbracket\cdot\rrbracket)\ (E,V) \triangleq \textbf{ if } \text{evts}(E) = \emptyset \textbf{ then } \emptyset \textbf{ else}\\ \quad \textbf{let } e = \text{first}(\text{evts}(E)) \textbf{ in}\\ \quad\quad \textbf{let } S = \llbracket e, V\rrbracket \textbf{ in}\\ \quad\quad\quad \textbf{let } S_R = \{R \mid \exists i.\, (i, R) \in S\} \textbf{ in}\\ \quad\quad\quad\quad S \cup \bigcup_{R \in S_R} \text{awk } (\llbracket\cdot\rrbracket)\ (\text{succ}(E, e),\ (V \cup R))\end{array}$$

Algorithm 1: awk function

This analysis is performed one event at a time (note that an event can change at most one value in memory). To keep the values which might get killed by a new event, at any given iteration, we do not only propagate the resulting relation R, but the union of this new result with the previous one V. This means that a future event will have access to the values of some previous result V computed during the analysis of the thread.

For example, the non-SC result $\{(\texttt{r1}, 0), (\texttt{r2}, 0)\}$ of Fig. 1 is not generated by octagon (written Oct in the following). Indeed, if we perform Oct on P_0, starting with $V = \{(x, 0), (y, 0), (\texttt{r1}, 0), (\texttt{r2}, 0)\}$, the only value that x can hold at line (b) is 1. Yet, the non-SC result is covered by awk(Oct) applied to the interleaving $(a), (b), (c), (d)$ of the code of Fig. 1. Indeed, if we apply awk(Oct)

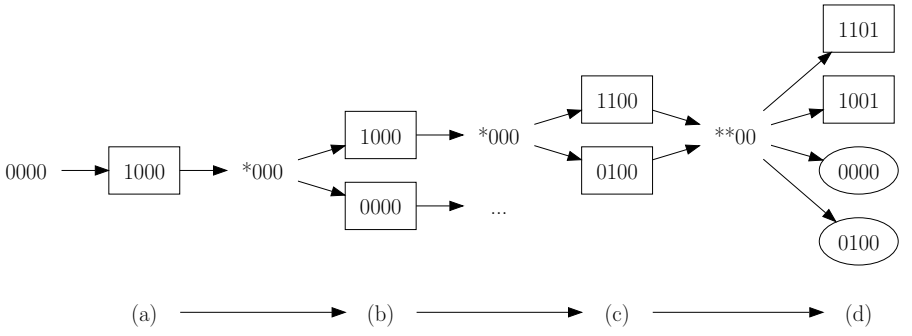


Fig. 5. Call graph of awk (Oct) ((*a*); (*b*); (*c*); (*d*), 0000)

with the same initial result V, we get $R = \{\{(x,1),(y,0),(\texttt{r1},0),(\texttt{r2},0)\}\}$ as the result of $\text{Oct}((a),V)$ for the first event. Then, we compute $V' = R \cup V$ and propagate it to line (*b*), *i.e.*, we compute $\text{Oct}((b),V')$. This means that the event at line (*b*) has access to the value $(x,0)$, for it appears in V'.

Fig. 5 gives a call graph of awk(Oct), on the interleaving $(a),(b),(c),(d)$ of the code of Fig. 1. A sequence $v_x v_y v_1 v_2$ represents $(x,y,\texttt{r1},\texttt{r2}) = (v_x,v_y,v_1,v_2)$. The non-framed sequences are the ones that we propagate at every call of awk(Oct), and the framed ones are the results that we return for every event.

Observe that we obtain the results 0000 and 0100 (in the ellipses) for event (*d*), both of them representing $(\texttt{r1},\texttt{r2}) = (0,0)$, *i.e.*, the non-SC behaviour. Indeed, when awk (Oct) $((d) : \texttt{r2} \leftarrow x, V)$ is executed, we not only have access to the previously computed result—as we would if we were directly applying Oct to the program—but rather to the union of all the previous results, including the initial one, where x holds 0.

Repair Loop. Finally, we define the repair loop as a transformation of an analysis of type $\mathcal{A}$ to a new analysis. The repair loop builds n_{SC} concatenations of the threads of E to simulate all the interleavings of E, then analyses them without killing any values, and finally takes the union of the results:

$$\text{repair-loop}([\![\cdot]\!])(E,V) \triangleq \bigcup_{E_{\text{big}} \in \text{concat}(E,n_{\text{SC}})} \text{awk}([\![\cdot]\!])(E_{\text{big}},V)$$

Revisiting the above example, applying repair-loop(Oct) to the code of Fig. 1 will simulate all the interleavings of the program. In particular, as we explained two paragraphs above, it simulates the interleaving $(a),(b),(c),(d)$ by the concatenation $P_0;P_0;P_1;P_1$. Then, it runs awk(Oct) on this concatenation, and since awk does not kill any value, we obtain the non-SC result $(\texttt{r1},\texttt{r2}) = (0,0)$, as explained in the preceding paragraph, and shown in Fig. 5.

5 Specification vs. Implementations of the Repair Loop

We gave in Sec. 4 a *specification* of the repair loop. Actual implementations are likely to use more scalable techniques such as fixed point computations. We leave the proof that such an implementation matches the specification of Sec. 4 for future work. Yet, we performed experiments to see how much an implementation of the repair loop would impair the performance of an analysis.

Experimental Cost of the Repair Loop. We showed that the points-to analysis of Rugina and Rinard is sound w.r.t. weak memory models even if performed with trace semantics, where the most precise results would be obtained. Yet, the analysis is implemented using a fixed point computation over the concurrent control flow graph instead of reasoning over all possible traces. As we showed above, the imprecision incurred by this fixed point iteration can be used to repair otherwise unsound analyses. A crucial question is, however, how many iterations are needed to arrive at a fixed point. We have shown that there exists a finite upper bound. It remains to be seen whether fewer iterations suffice in practice.

A similar study of the cost of such a fixed point iteration has recently been undertaken by Miné [16], and our experiments confirm his results: we use three sets of benchmarks that were previously used as case studies on the analysis of concurrent software: (1) concurrency bug patterns from the Apache web server as used in [26] (atom001, atom001a, atom002, atom002a, banking/av, banking/no_av, banking/some_av), (2) the banking and indexer examples from [27] (banking and indexer), and (3) several Linux device drivers together with nondeterministic environments as generated by DDVerify [28]. The detailed results and the source code of all experiments together with our implementation of Rugina and Rinard's points-to analysis is available at `http://www.cprover.org/wmm/`.

For several benchmarks of different origin our results confirm the findings of Miné: the typical number of iterations to reach a fixed point is very low, in fact it is always 2 in our samples. To study the overhead of repair iterations we summed up the time spent in all but the first iteration for each thread. We observe that this time overhead is very small with at most 0.034 seconds.

6 Related Work and Conclusion

We refer the reader to [29,30] for an overview of the issues related to weak memory models. Program analyses for concurrent programs running on weak architectures have recently been considered by Ferrara [15] and Miné [16].

To the best of our knowledge, however, there is no general result on the soundness of program analyses for weak memory models. Both Ferrara and Miné describe extensions of the abstract interpretation framework to concurrent programs. In contrast to our work, which is generic, [15] explicitly focuses on an over-approximation of the Java memory model. Soundness w.r.t. the memory model is achieved by a fixed point iteration that implements the repair loop described in our paper. Miné [16] describes an extension of abstract interpretation to programs with a fixed number of threads and shared memory. He uses

Analysis	Soundness w.r.t. weak memory models
Knoop *et al.* [33]	yes (separable)
Chugh *et al.* [34]	yes (if no datarace)
Steensgaard [35]	yes (flow-insensitive)
Miné [16]	yes
Rugina and Rinard [24]	yes
Jeannet [14]	no
Ferrara [15]	yes on Java Memory Model
Farzan and Kincaid [25]	yes (separable)
Khedker and Dhamdhere [36]	separable: yes; non-separable: not in general
Constant propagation [32]	yes (non-relational)

Fig. 6. Soundness of some concurrent analyses w.r.t. weak memory models

a fixed point iteration that is similar to the approach described by Rugina and Rinard [24] to compute a safe over-approximation of all possible interleavings. Furthermore Miné proves these results to be sound for a class of weak memory models specified as program transformations. Unlike our work, which is based on a framework for weak memory models that provably embraces several existing models, the modelling power of these transformations is unclear. With the results presented in the present paper it follows immediately that Miné's analysis extends from sequential consistency to weak memory models by the repair loop.

Sevcik and Vafeiadis *et al.* [18,19] prove the correctness of a compiler for concurrent C programs towards x86 assembly, targeting the TSO model of [31]. Thus, they have to prove that analyses such as *constant propagation* [32] preserve the semantics from the source to the target program, which requires proving properties similar to ours. Since constant propagation is non-relational, we not only showed its soundness for TSO, but also for a large class of other models.

Several other analyses have been extended from sequential programs to the concurrent setting without explicitly discussing the effect of weak memory models. In the following, we survey their soundness w.r.t. weak memory models in the light of the results presented here. We summarise this discussion in Fig. 6.

We already discussed Rugina and Rinard's analysis [24] in Sec. 3.

Jeannet [14] presents the stack abstraction underlying ConcurInterproc, which can be combined with data abstraction domains such as octagons [13] or convex polyhedra [37] to apply abstract interpretation to parallel programs with a fixed number of threads. This approach is not generally sound for weak memory models as shown in the introduction.

Khedker and Dhamdhere [36] give a definition of separability for data flow analyses, *i.e.*, analyses where each data flow fact may be tracked in isolation, independently of the valuations of other data flow facts. Although separability is a concept independent of an analysis being (non-)relational, all separable problems can be expressed with the type that we proved to be sound in Thm. 1. For non-separable problems, we are unable to make such a general statement.

Most notably, bit-vector analyses are separable data flow analysis problems. Therefore the approaches described by Knoop *et al.* [33] and Farzan and Kincaid [25], who present methods of adapting a unidirectional bit-vector analysis

designed with sequential programs in mind for use with multi-threaded programs, are immediately sound for weak memory models if sound for SC.

As another well-established classification of analyses consider flow sensitive vs. flow insensitive analyses. Rinard observes in [11] that flow insensitive analyses such as Steensgaard's pointer analysis [35] are SC-sound. Hence by Thm. 1, we conclude that they are also sound for weak memory models.

The provable soundness of both bit-vector analyses and flow insensitive analyses is of uttermost practical importance as analyses of these kinds are used in optimizing compilers. Although today's compilers do not yet implement optimizations for multi-threaded programs in a concurrency aware fashion, our results show that it would be safe to add such extensions.

Future Work. While we already have a strong result for non-relational analyses, we would like to further refine our results for relational ones, *e.g.*, by lifting the restriction on the analysis being forward. Moreover, we intend to exercise our specification of the repair loop as given in Sec. 4, by proving that existing implementations, *e.g.*, Rugina and Rinard's, actually satisfy this property.

Acknowledgements. We thank Vijay D'Silva, Peter Sewell, Viktor Vafeiadis and Thomas Wahl for invaluable discussions and comments.

References

1. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1979)
2. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 30. (March 2009), intel.com/products/processor/manuals
3. IBM: Power ISA Version 2.06B (July 2010), power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf
4. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 258–272. Springer, Heidelberg (2010)
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: Running Tests Against Hardware. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 41–44. Springer, Heidelberg (2011)
6. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: POPL (2005)
7. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI (2008)
8. Adve, S.V., Hill, M.D.: Weak ordering – A new definition. In: ISCA (1990)
9. Burckhardt, S., Alur, R., Martin, M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
10. Alglave, J., Maranget, L.: Stability in Weak Memory Models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
11. Rinard, M.: Analysis of Multithreaded Programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 1–19. Springer, Heidelberg (2001)
12. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: International Symposium on Programming, Dunod (1976)

13. Miné, A.: The octagon abstract domain. In: Workshop on Analysis, Slicing, and Transformation (AST). IEEE (2001)
14. Jeannet, B.: Relational interprocedural verification of concurrent programs. In: SEFM. IEEE (2009)
15. Ferrara, P.: Static Analysis Via Abstract Interpretation of the Happens-before Memory Model. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 116–133. Springer, Heidelberg (2008)
16. Miné, A.: Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 398–418. Springer, Heidelberg (2011)
17. Alglave, J.: A Shared Memory Poetics. PhD thesis, Université Paris 7 and INRIA (2010), http://moscova.inria.fr/~alglave/these
18. Sevcik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL (2011)
19. Vafeiadis, V., Zappa Nardelli, F.: Verifying Fence Elimination Optimisations. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 146–162. Springer, Heidelberg (2011)
20. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.: x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors. In: CACM (2010)
21. SPARC: SPARC Architecture Manual Versions 8 and 9 (1992 and 1994), sparc.org/standards/V8.pdf, sparc.org/standards/SPARCV9.pdf
22. Compaq: Alpha Architecture Reference Manual, 4 edn. (2002), download.majix.org/dec/alpha_arch_ref.pdf
23. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus (1999)
24. Rugina, R., Rinard, M.C.: Pointer analysis for multithreaded programs. In: PLDI (1999)
25. Farzan, A., Kincaid, Z.: Compositional Bitvector Analysis for Concurrent Programs with Nested Locks. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 253–270. Springer, Heidelberg (2010)
26. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-Based Symbolic Analysis for Atomicity Violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
27. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic Predictive Analysis for Concurrent Programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)
28. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: ASE. ACM (2007)
29. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29, 66–76 (1995)
30. Adve, S., Boehm, H.J.: Memory Models: A Case for Rethinking Parallel Languages and Hardware. To appear in CACM
31. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
32. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural constant propagation. In: SIGPLAN Symposium on Compiler Construction (1986)
33. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst. 18(3), 268–299 (1996)

34. Chugh, R., Voung, J.W., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: Programming Language Design and Implementation (PLDI), pp. 316–326. ACM (2008)
35. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL (1996)
36. Khedker, U.P., Dhamdhere, D.M.: A generalized theory of bit vector data flow analysis. ACM Trans. Program. Lang. Syst. 16(5), 1472–1511 (1994)
37. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)

Towards a General Theory of Barbs, Contexts and Labels*

Filippo Bonchi[1], Fabio Gadducci[2], and Giacoma Valentina Monreale[2]

[1] ENS Lyon, Université de Lyon, LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)
[2] Dipartimento di Informatica, Università di Pisa

Abstract. Barbed bisimilarity is a widely-used behavioural equivalence for interactive systems: given a set of predicates (denoted "barbs", and representing basic observations on states) and a set of contexts (representing the possible execution environments), two systems are deemed to be equivalent if they verify the same barbs whenever inserted inside any of the chosen contexts. Despite its flexibility, this definition of equivalence is unsatisfactory, since often the quantification is over an infinite set of contexts, thus making barbed bisimilarity very hard to be verified.

Should a labelled operational semantics be available for our system, more efficient observational equivalences might be adopted. To this end, a series of techniques have been proposed to derive labelled transition systems (LTSs) from unlabelled ones, the main example being Leifer and Milner's reactive systems. The underlying intuition is that labels are the "minimal" contexts that allow for a reduction to be performed.

We introduce a framework that characterizes (weak) barbed bisimilarity via transition systems whose labels are (possibly minimal) contexts. Differently from other proposals, our theory is not dependent on the way LTSs are built, and it relies on a simple set-theoretical presentation. To provide a test-bed for our formalism, we instantiate it by addressing the semantics of mobile ambients and HoCore, recasting the (weak) barbed bisimilarities of these calculi via label-based behavioural equivalences.

Keywords: Barbed bisimilarity, contexts as labels, reactive systems.

1 Introduction

The operational semantics of process calculi was usually given in terms of labelled transition systems (LTSs), i.e., a set of possible states, plus a labelled transition relation among them, describing the possible evolutions of the computation. The labels express some kind of basic statement about the evolutions themselves, and thus they allow for an easy and fruitful way to provide meaningful behavioural semantics based on observations, i.e., basically looking at the labels of the state evolutions available for a calculus.

* The first author has been supported by CNRS PEPS CoGIP. The second and third author have been partially supported by the EU FP7-ICT IP ASCEns and by the MIUR PRIN SisteR (PRIN 20088HXMYN).

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 289–304, 2011.

More recently, however, the growing syntactical complexity of these calculi made almost customary to present their behaviour by a reduction semantics: an unlabelled relation, defined modulo a congruence that equates those processes intuitively representing the same system specification. This paradigmatic shift stimulated the adoption of *barbed equivalences* [13]: behavioural semantics based on a family of state predicates, called *barbs*, intended to capture the ability of a process of performing an interaction with the environment.

The trade-off to the relative easiness in defining barbed congruences even for operationally rich calculi is given by the difficulty of formally checking them: indeed, the verification usually requires to evaluate the barbs of a system with respect to all the contexts it can be possibily inserted in. Hence, the ingenuity of the researcher focussed on devising suitable labelled semantics from reduction ones, cases at hand being the calculus of mobile ambients (MAs) [7] and the core calculus for higher-order concurrency (HoCore) [10]. Indeed, a series of techniques [9,17,8,1,16] have been proposed to address the automatic derivation of an LTS starting from a reduction semantics, in order to distill behavioural equivalences that are congruences. Among these proposals, the most renowned one is Leifer and Milner's theory of reactive systems (RSs): the underlying intuition is that the labels of the derived transition system (called IPO LTS) are the "minimal" contexts that allow for a reduction to be performed, where minimality is captured by the categorical notion of relative pushout [11].

Our paper moves from Leifer and Milner's seminal work: our aim is to identify suitable conditions under which (weak) barbed bisimilarity can be characterized in terms of a behavioural equivalence over a suitably labelled transition system. Differently from the original theory of RSs, though, as well as from more recent contributions, the present work does not focus on devising novel techiques for distilling an adequate LTS, possibly for open systems [9], or for identifying what the "right" notion of barbs should be [14]: on the contrary, it tries and identifies a class of transition systems (with contexts as labels) that may represent a meaningful abstraction of an underlying reduction semantics. Here, we assume that "meaningful abstraction" precisely means that we may recast barbed bisimilarity using the bisimulation game in the LTS.

To this end, we introduce context LTSs, a general notion of which IPO LTSs are an instance. We then present weak context bisimilarity as the equivalence for these LTSs adopting the standard bisimulation game, and provide conditions ensuring that it is a congruence. We also use contexts as labels to define weak L-bisimilarity. It adopts a bisimulation game that is asymmetric with respect to a set L of contexts, yet it is a congruence and, depending on L, it may represent an efficient characterisation for barbed bisimilarity, which avoids to consider all contexts. Finally, in order to properly establish the adequacy of our theory, we check it against suitable case studies. So, we instantiate our proposal over MAs and HoCore, addressing their weak semantics: the former has a notably complex barbed bisimilarity, resilient until recently to a labelled characterization [12]; and only the strong semantics was so far considered for the latter. Their complementary features allow for testing the expressiveness of our framework.

Related Works. Our theory should be considered as an outcome of the stream of research born out of Leifer and Milner's theory [11], part of the contexts as labels approach for LTSs. However, the emphasis is not on the procedure for distilling the right class of contexts to be chosen as labels, but in the identification of a family of LTSs that are able to characterize barbed bisimilarity. Thus, our sound and complete context LTSs (Definition 10) are a general notion of which IPO LTSs are an instance [11], in the same way that Definition 13 subsumes the property of *having redex RPOs*. Similarly, Theorem 2 extends the congruence result for RSs [11, Theorem 1] to the new setting and for the weak semantics. The notion of (weak) barbed semantics for RSs was presented in [4], and strong L-bisimilarity discussed in [3]. Properties (1) and (2) of Definition 10 were exploited in [5] under the name of soundness and completeness for an LTS.

Synopsis. § 2 summarizes the main notions concerning our case studies, namely MAs (§ 2.1) and HoCore (§ 2.2). § 3 introduces our framework: the standard notions of RSs (albeit recast in a novel, set-theoretical way) and of weak barbed semantics (with a new result on non-discriminating contexts, § 3.1), while the labelled semantics that we propose (weak context bisimilarity, § 3.2) exploits instead the original notion of sound and complete context LTS. § 4 proposes weak L-bisimilarity, the proof that (under mild conditions on L) it is a congruence, and its correspondence with weak barbed semantics. § 5 and 6 show how our theory captures weak bisimilarity for MAs and HoCore, respectively. Finally, § 7 draws some conclusions and outlines directions for further research.

2 Two Case Studies

2.1 Mobile Ambients

This section introduces the finite, communication-free fragment of Mobile Ambients (MAs): its reduction semantics and behavioural equivalence [7], and the labelled transition system (LTS) for the calculus proposed in [2].

The syntax is shown in Fig. 1(a). We assume a set $\mathcal{N}$ of *names* ranged over by $m, n, o, \ldots$ and we let $P, Q, R, \ldots$ range over the set $\mathcal{P}_M$ of processes. The *free names* of a process P (denoted by $fn(P)$) are defined as usual. Processes are taken up to a *structural congruence*, axiomatised in Fig. 1(b) and denoted by $\equiv$. The *reduction relation* $\rightsquigarrow_M$, describes process evolution: it is the least relation $\rightsquigarrow_M : \mathcal{P}_M \times \mathcal{P}_M$ closed under $\equiv$ and generated by the rules in Fig. 1(c).

A *strong barb* o is a predicate over processes, with $P \downarrow_o$ denoting that P satisfies o. In MAs, $P \downarrow_n$ denotes the presence at top-level of an unrestricted ambient n. Formally, $P \downarrow_n$ if $P \equiv (\nu A)(n[Q]|R)$ and $n \notin A$, for some processes Q and R and a set of restricted names A. A process P satisfies the *weak barb* n (denoted as $P \Downarrow_n$) if there exists a process P' such that $P \rightsquigarrow_M^* P'$ and $P' \downarrow_n$, where $\rightsquigarrow_M^*$ is the transitive and reflexive closure of $\rightsquigarrow_M$.

Strong and weak barbs are exploited to define the standard equivalence for MAs: *weak reduction barbed congruence*, of which a labelled characterization is in [12]. Before presenting it, we introduce MAs *contexts*: they are MAs processes with a hole $-$, formally generated by the following grammar (for $R \in \mathcal{P}_M$)

(a) $P ::= \mathbf{0}, n[P], M.P, (\nu n)P, P_1|P_2 \qquad M ::= in\ n, out\ n, open\ n$

(b)

$$P|Q \equiv Q|P \qquad (\nu n)(P|Q) \equiv P|(\nu n)Q \quad \text{if } n \notin fn(P)$$
$$(P|Q)|R \equiv P|(Q|R) \qquad (\nu n)m[P] \equiv m[(\nu n)P] \quad \text{if } n \neq m$$
$$P|\mathbf{0} \equiv P \qquad (\nu n)M.P \equiv M.(\nu n)P \quad \text{if } n \notin fn(M)$$
$$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P \qquad (\nu n)P \equiv (\nu m)(P\{^m/_n\}) \quad \text{if } m \notin fn(P)$$

(c)

$$n[in\ m.P|Q]|m[R] \rightsquigarrow_M m[n[P|Q]|R] \qquad \text{if } P \rightsquigarrow_M Q \text{ then } (\nu n)P \rightsquigarrow_M (\nu n)Q$$
$$m[n[out\ m.P|Q]|R] \rightsquigarrow_M n[P|Q]|m[R] \qquad \text{if } P \rightsquigarrow_M Q \text{ then } n[P] \rightsquigarrow_M n[Q]$$
$$open\ n.P|n[Q] \rightsquigarrow_M P|Q \qquad \text{if } P \rightsquigarrow_M Q \text{ then } P|R \rightsquigarrow_M Q|R$$

Fig. 1. (a) Syntax, (b) structural congruence and (c) reduction relation of MAs

$$C[-] ::= -,\ n[C[-]],\ M.C[-],\ (\nu n)C[-],\ C[-] \mid R.$$

Definition 1 (Weak Reduction Barbed Congruence). Weak reduction barbed congruence $\approx_M$ *is the largest symmetric relation* $\mathcal{B}$ *s.t.* $P\,\mathcal{B}\,Q$ *implies*

- *if* $P \downarrow_n$ *then* $Q \Downarrow_n$*;*
- *if* $P \rightsquigarrow_M P'$ *then* $Q \rightsquigarrow_M^* Q'$ *and* $P'\,\mathcal{B}\,Q'$*;*
- $\forall C[-], C[P]\,\mathcal{B}\,C[Q]$.

An LTS for MAs. In Fig. 2 we present the LTS M for MAs proposed in [2], and obtained by a suitable application of Leifer and Milner's theory [17] (hence, its transition labels are the "minimal" contexts allowing a reduction to occur). Note that we assume that the LTS is closed with respect to structural congruence.

The rule Tau represents the τ-actions modeling reduction of the process. The rule ExtRed represents three rules, one for each axiom of the reduction relation (see Fig. 1(c), left): the process L of the label represents the left hand side of the axiom and the process R of the target state represents the right hand side (thus, ExtRed model the reduction performed by the environment).

The other rules instead model the interactions of a process with its environment. Note that in the conclusions of some rules there are names and processes (denoted by o, S_1, S_2) that do not appear in the premises. These represent ambient names and processes that are provided by the environment and thus we must always assume that $(\{o\} \cup fn(S_1) \cup fn(S_2)) \cap A = \emptyset$. For instance, the rule Open enables a proces to open an ambient provided by the environment, while the rule CoOpen allows the environment to open an ambient of the process. For a description of the other rules, we refer the reader to [2].

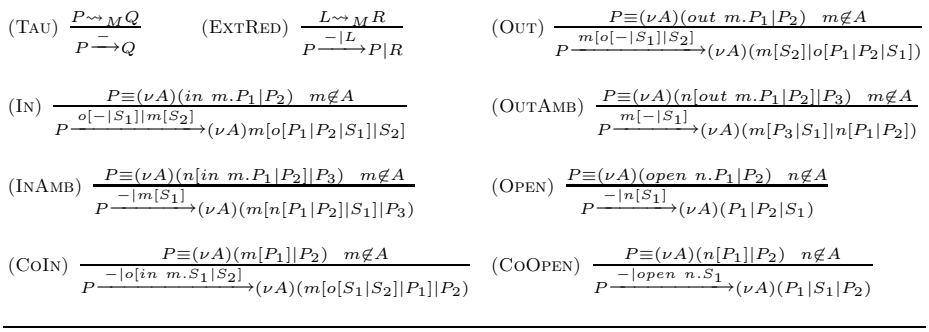


Fig. 2. The LTS M, for $L \rightsquigarrow_M R$ ranging over the three axioms of $\rightsquigarrow_M$

2.2 HoCore

HoCore [10] has been introduced as a core language for higher-order concurrency (where processes may exchange messages containing processes). Its main peculiarities consist in the fact that (1) several different notions of strong behavioural equivalence coincide, and (2) they are all computable, even if the formalism is Turing complete. We consider here weak equivalence: it was not studied before, even if it is immediate to note that (1) does not hold anymore.

The syntax is shown in Fig. 3(a). We assume a set $\mathcal{N}$ of *names* ranged over by $a, b, c, \ldots$ and a set $\mathcal{V}$ of *process variables* ranged over by $x, y, z, \ldots$, requiring $\mathcal{N}$ and $\mathcal{V}$ to be disjoint. The set of names of a process P (denoted by $n(P)$) is defined as usual. The input $a(x).P$ binds the free occurrences of the variable x in P. We write $fv(P)$ for the set of free variables of P, and we identify processes up-to the renaming of bound variables. A process is *closed* if it does not have free variables. To make the presentation lighter, we say *process* to mean closed process, and we write *open process* whenever we mean process that might not be closed. We let $P, Q, R, \ldots$ range over the set $\mathcal{P}_H$ of closed processes.

HoCore *contexts* are closed processes with a hole $-$, formally generated by the following grammar (for $Q \in \mathcal{P}_H$)

$$C[-] ::= -,\ a(x).C[-],\ C[-] \mid Q.$$

The structural congruence $\equiv$ is the smallest congruence induced by the axioms in Fig. 3(b). The behaviour of a process P is then described as the reaction relation $\rightsquigarrow_H$ over processes up to $\equiv$, obtained by the rules in Fig. 3(c).

Barbs are defined as follows: $P \downarrow_{\bar{a}}$ if $P \equiv \overline{a}P_1|P_2$, for processes P_1 and P_2. A process P satisfies the *weak barb* $\bar{a}$ (denoted as $P \Downarrow_{\bar{a}}$) if there exists a process P' such that $P \rightsquigarrow_H^* P'$ and $P' \downarrow_{\bar{a}}$, where $\rightsquigarrow_H^*$ is the transitive and reflexive closure of $\rightsquigarrow_H$. The notion of barb is used in [10] to give the definition of *asynchronous barbed congruence*. Here we straightforwardly extend it to the weak case.

Definition 2 (Weak Asynchronous Barbed Congruence). Weak asynchronous barbed congruence $\sim_H^B$ *is the largest symmetric relation* $\mathcal{B}$ *s.t.* $P\,\mathcal{B}\,Q$ *implies*

(a) $P ::= \mathbf{0},\ \overline{a}P,\ a(x).P,\ x,\ P_1|P_2$ (b) $P|\mathbf{0} \equiv P\ (P|Q)|R \equiv P|(Q|R)\ P|Q \equiv Q|P$

(c) $\overline{a}Q|a(x).P \rightsquigarrow_H P\{^Q/_x\}$ if $P \rightsquigarrow_H Q$ then $P|R \rightsquigarrow_H Q|R$

Fig. 3. (a) Syntax, (b) structural congruence and (c) reduction relation of HoCore

- *if* $P \Downarrow_{\bar{a}}$ *then* $Q \Downarrow_{\bar{a}}$*;*
- *if* $P \rightsquigarrow_H^* P'$ *then* $Q \rightsquigarrow_H^* Q'$ *and* $P' \mathcal{B} Q'$*;*
- $\forall C[-], C[P] \mathcal{B} C[Q]$.

For instance, $a(x).\overline{a}x \sim_H^B \mathbf{0}$. This is intuitively understood by observing that the former process can interact only with a context of the shape $-|\overline{a}P_1|P_2$. Moreover, the interaction of $a(x).\overline{a}x$ with $-|\overline{a}P_1|P_2$ reduces to the process $\overline{a}P_1|P_2$, which is clearly equivalent to $\mathbf{0}|\overline{a}P_1|P_2$ (i.e., $\mathbf{0}$ inside the context $-|\overline{a}P_1|P_2$).

Concerning the equalities above, it is worth noting that $a(x).\overline{a}x$ and $\mathbf{0}$ are not equivalent with respect to the *strong* asynchronous barbed congruence (obtained by replacing $\Downarrow_{\bar{a}}$ with $\downarrow_{\bar{a}}$ and $\rightsquigarrow^*$ by $\rightsquigarrow$ in the above definition).

Two LTSs for HoCore. Fig. 4 shows the LTS semantics of HoCore introduced in [10] for open processes (the symmetric variants of the two rightmost rules are omitted). We let α denote any label of a transition and $bv(\alpha)$ the set of its bound variables (for $bv(a(x)) = \{x\}$ and $bv(\alpha) = \emptyset$ for $\alpha \neq a(x)$).

This LTS is used in [10] to define a few alternative notions of equivalence that (when restricted to closed processes) are shown to coincide with *strong* asynchronous barbed congruence. What is noteworthy is that these equivalences avoid the quantification over all contexts of asynchronous barbed congruence, and they are thus proved to be computable.

These behavioural equivalences (making use of the LTS) cannot be straightforwardly extended to the weak case, and indeed their "naive weak extension" would not coincide with $\sim_H^B$. This can be straightforwardly observed by noting that all the definitions in [10] are "synchronous", i.e., they require that every input transition is matched by another input transition. On the contrary, any equivalence defined in such a manner would never equate the processes $a(x).\overline{a}x$ and $\mathbf{0}$, which instead are related, as discussed before, by $\sim_H^B$.

At the end of this paper, we will apply our theoretical framework to derive a labelled characterization of $\sim_H^B$. We will make use of the LTS H shown in Fig. 5 where, once more, labels are (minimal) contexts, as in the LTS for MAs in Fig. 2.

The Tau rule models internal computations. The In rule models the communication over a channel a, where the environment send a process S to the process P. Vice versa, in the Out rule the process P send a process P_1 to the the environment: the sent process P_1 is substituted in the continuation S of the inputting environment. The rule ExtRed models transitions where the redex is fully offered by the environment. It represents the axiom of the reduction relation (see Fig. 3(c), left): the process L of the label and the process R

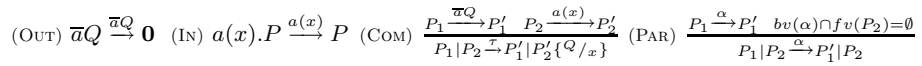


Fig. 4. The ordinary LTS of HoCore

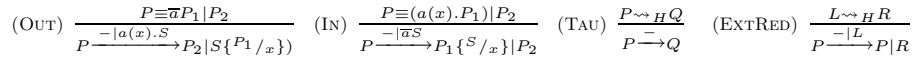


Fig. 5. The LTS H, for $L \rightsquigarrow_H R$ ranging over the three axioms of $\rightsquigarrow_H$

of the target state represent the left-hand and the right-hand side of the axiom, respectively.

Even if we do not state it formally, it is immediate to note the tight correspondence between the ordinary LTS (Fig. 4) and our own H, modulo the structural congruence: if $P \xrightarrow{\tau} Q$ then $P \xrightarrow{-} Q$, and vice versa if $P \xrightarrow{-} Q$ there exist P', Q' s.t. $P' \xrightarrow{\tau} Q'$ and $P \equiv P'$, $Q \equiv Q'$ (and similarly for $P \xrightarrow{a(x)} Q$ and $P \xrightarrow{\overline{a}P_1} Q$ with respect to $P \xrightarrow{-|\overline{a}S} Q\{^S/_x\}$ and $P \xrightarrow{-|a(x).S} Q|S\{^{P_1}/_x\}$, respectively).

Remark 1. It is worth remarking here that the ordinary LTS is finite, while H is not because S represents any possible process provided by the environment. This kind of problem occurs also for the LTS M that we have shown for MAs and for other LTSs that have been proposed in analogous works (see e.g. [15,16]). It is challenging to devise a general solution to this problem, but some suggestions come from *open* reactive systems [9]: instead of considering only closed process and contexts, one might take into account *open* processes, contexts and *variable substituions*. In the special case of HoCore, it would be interesting to develop (for the weak case) a technique analogous to the *normal bisimulation* of [10]: instead of considering all the possible S in the transitions $P \xrightarrow{-|\overline{a}S}$ and $P \xrightarrow{-|a(x).S}$, we might take only $S = \overline{m}\mathbf{0}$ and $S = m(y).x$, respectively, for a fresh name m. This is out of the scope of the present paper and is left for future work.

3 Reactive Systems and Context LTSs

This section introduces a framework that encompasses MAs and HoCore, aiming at a general theory for modeling the weak (barbed) semantics of interactive formalisms. After providing the (set-theoretical) definition of reactive system, we introduce a barbed (§ 3.1) and a labelled semantics for these systems (§ 3.2), showing that the latter is, under suitable conditions, a congruence (§ 3.3).

We first define a notion of *system theory*, denoting how the states of a system are built. Recall that a monoid $\mathbb{M} = (\mathcal{M}, \otimes, 1)$ consists of a set $\mathcal{M}$, an associative binary operator $\otimes : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, and the identity element $1 \in \mathcal{M}$. Given a

monoid $\mathbb{M}$ and a set X, a monoid action of $\mathbb{M}$ on X is an operation $\cdot : \mathcal{M} \times X \to X$ compatible with the monoid operation, i.e., such that for each $m_1, m_2 \in \mathcal{M}$ and $x \in X$, $m_1 \cdot (m_2 \cdot x) = (m_1 \otimes m_2) \cdot x$ and for each $x \in X$, $1 \cdot x = x$.

Definition 3 (System Theory). *A* system theory *is a triple* $\mathbb{S} = \langle \mathcal{P}, \mathbb{C}, \cdot \rangle$ *such that* $\mathcal{P}$ *is a set of* processes*, ranged over by* $P, Q, R, \ldots$, $\mathbb{C} = (\mathcal{C}, \circ, -)$ *a monoid of* contexts*, ranged over by* $C[-], C_1[-], \ldots$, *and* $\cdot : \mathcal{C} \times \mathcal{P} \to \mathcal{P}$ *a monoid action.*

We usually denote context composition $C_1[-] \circ C_2[-]$ as $C_2[C_1[-]]$ and the action $C[-] \cdot P$ as $C[P]$. The chosen notation supports the intuition that the monoid operation represents the functional composition of unary contexts, while the action is just the insertion of a process into a context. It allows for an easier comparison with the process calculi notation adopted in later sections.

Definition 4 (Reactive System). *A* reactive system *(RS) is a triple* $\mathcal{R} = \langle \mathbb{S}, \rightsquigarrow, O \rangle$ *where* $\mathbb{S}$ *is a system theory,* $\rightsquigarrow \subseteq \mathcal{P} \times \mathcal{P}$ *a transition relation and* O *a set of predicates on* $\mathcal{P}$*.*

We write $P \rightsquigarrow Q$ to mean $\langle P, Q \rangle \in \rightsquigarrow$, and we denote by $\rightsquigarrow^*$ the reflexive and transitive closure of $\rightsquigarrow$. The predicates in O are called *barbs* and they represent basic observations on the state of a system. We write $P \downarrow_o$ if P satisfies $o \in O$. Analogously, the state P satisfies the weak barb o (written $P \Downarrow_o$) if there exists a state P' such that $P \rightsquigarrow^* P'$ and $P' \downarrow_o$.

3.1 Barbed Saturated Semantics

With the ingredients offered by our theory, we can define a behavioural equivalence which equates two processes if these cannot be distinguished by an observer that can insert a process into any context and check the exposed barbs.

In the following, we fix an RS $\mathcal{R} = \langle \mathbb{S}, \rightsquigarrow, O \rangle$, with $\mathbb{S} = \langle \mathcal{P}, \mathbb{C}, \cdot \rangle$.

Definition 5 (Weak Barbed Saturated Bisimilarity). Weak barbed saturated bisimilarity $\approx^{BS}$ *(for* $\mathcal{R}$*) is the largest symmetric relation* $\mathcal{B}$ *such that* $P \mathcal{B} Q$ *implies* $\forall C[-] \in \mathcal{C}$

- *if* $C[P] \Downarrow_o$ *then* $C[Q] \Downarrow_o$*;*
- *if* $C[P] \rightsquigarrow^* P'$ *then* $C[Q] \rightsquigarrow^* Q'$ *and* $P' \mathcal{B} Q'$*.*

Weak barbed saturated bisimilarity (which is a congruence) is general enough to encompass the standard behavioural equivalences of many process calculi. The main drawback of this kind of definition is the quantification over all contexts that makes hard the proofs of equivalence. In this paper, we provide a general proof technique for $\approx^{BS}$ that avoids the quantification over all possible contexts by relying on LTSs like those derived by Leifer and Milner's theory [11].

It is sometimes possible to restrict the set of contexts to be checked against, so obtaining again the same barbed equivalence. This can be accomplished by identifying those contexts whose presence does not really influence $\approx^{BS}$.

Definition 6 (Non-Discriminating Context). *Let $E[-] \in \mathcal{C}$ be a context. It is* non-discriminating *(for $\mathcal{R}$) if $\forall P \in \mathcal{P}$, $\forall C[-] \in \mathcal{C}$ and $\forall o \in \mathcal{O}$*

1. *if $C[E[P]] \downarrow_o$ then $\forall Q \in \mathcal{P}.\ C[E[Q]] \downarrow_o$;*
2. *if $E[P] \rightsquigarrow P'$ then $P' = E'[P]$ and $\forall Q \in \mathcal{P}.\ E[Q] \rightsquigarrow E'[Q]$.*

For instance, the context $M.-$ of MAs is non-discriminating, since it hides the strong barbs of the process that is inserted inside it, and it inhibits its transitions. For the same reason, the context $a(x).-$ of HoCore is non-discriminating as long as it is applied to closed processes (thus, no variable is actually bound).

Now, starting from an RS $\mathcal{R}$ we can build a new reactive system $\mathcal{R}'$ by removing some (possibly all) non-discriminating contexts. The following proposition ensures that the barbed saturated bisimilarity in the two systems coincide.

Proposition 1. *Let $\mathbb{C}'$ be a submonoid of the context monoid $\mathbb{C}$, $\mathbb{S}' = \langle \mathcal{P}, \mathbb{C}', \cdot \rangle$ the system theory derived from $\mathbb{S}$, and $\mathcal{R}' = \langle \mathbb{S}', \rightsquigarrow, O \rangle$ the RS derived from $\mathcal{R}$. If all contexts in $\mathbb{C}$ but not in $\mathbb{C}'$ are non-discriminating in $\mathcal{R}$, then $\approx^{BS}_{\mathcal{R}} = \approx^{BS}_{\mathcal{R}'}$.*

3.2 Weak Context Bisimilarity

In order to equip RSs with a labelled equivalence, we need to introduce a more basic notion, identifying those contexts that always allow a reaction.

Definition 7 (Reactive Context). *Let $C[-] \in \mathcal{C}$ be a context. It is* reactive *(for $\mathcal{R}$) if $\forall P \in \mathcal{P}$, if $P \rightsquigarrow P'$ then $C[P] \rightsquigarrow C[P']$. The set (actually, submonoid) of reactive contexts is denoted $\mathbb{R}$.*

For instance, the contexts $-|R$ and $(\nu a)-$ of MAs and $-|R$ of HoCore are reactive, while the prefixes $(a(x).-$ in HoCore and $M.-$ in MAs) are not.

Definition 8 (Context LTS). *A* context *LTS is a triple $\mathbb{D} = \langle \mathcal{R}, \mathcal{D}, \rightarrow_D \rangle$ such that $\mathcal{D} \subseteq \mathcal{C}$ and $\rightarrow_D \subseteq \mathcal{P} \times \mathcal{D} \times \mathcal{P}$.*

As usual, we write $P \xrightarrow{C[-]}_D Q$ to mean that $\langle P, C[-], Q \rangle \in \rightarrow_D$. Note moreover that the set of labels $\mathcal{D}$ is a subset of the set of all contexts $\mathcal{C}$. For instance, the LTSs M and H (in Figs. 2 and 5) are two context LTSs. The set of labels of H is $\{-, -|\overline{a}S, -|a(x).S \mid a \in \mathcal{N} \text{ and } S \in \mathcal{P}_H\}$.

In order to characterize the class of LTSs on which the labelled semantics for RSs is based, we need to introduce the following definition.

Definition 9 (Decomposition Pair). *Let $\langle C_1[-], C_2[-] \rangle$ a pair belonging to $\mathcal{D} \times \mathbb{R}$. It is a decomposition pair for $C[P] \rightsquigarrow P'$ if there exists a process P'' such that $P \xrightarrow{C_1[-]}_D P''$, $C_2[C_1[-]] = C[-]$ and $C_2[P''] = P'$.*

For instance, consider the MAs process $P = \mathit{open}\, n.P_1$ and the context $C[-] = -|n[P_2]|P_3$. A decomposition pair for $C[P] \rightsquigarrow_M P_1|P_2|P_3$ is $\langle -|n[P_2], -|P_3 \rangle$. Indeed, it is easy to check that $P \xrightarrow{-|n[P_2]} P_1|P_2$; moreover, if we compose $-|n[P_2]$

with $-|P_3$ we exactly obtain $C[-]$, and finally the composition between $P_1|P_2$ and $-|P_3$ gives $P_1|P_2|P_3$.

Now we can characterize the class of LTSs we are interested in: they are context LTSs satisfying suitable soundness and completeness properties.

Definition 10 (Sound and Complete Context LTS). *A context LTS is* sound *and* complete *if it satifies the properties*

1. *if* $P \xrightarrow{C[-]}_D P'$ *then* $C[P] \rightsquigarrow P'$;
2. *each* $C[P] \rightsquigarrow P'$ *has a decomposition pair* $\langle C_1[-], C_2[-]\rangle$.

The LTS M for MAs and the LTS H for HoCore are both sound and complete, as we discuss later in § 5 and § 6, respectively.

In the following, we fix a sound and complete context LTS $\mathbb{D}$. Moreover, we use $P \xrightarrow{C[-]} *_D P'$ to denote $P \rightsquigarrow^* \bullet \xrightarrow{C[-]}_D \bullet \rightsquigarrow^* P'$.

Definition 11 (Weak Context Bisimilarity). Weak context bisimilarity $\approx^D$ *(for* $\mathbb{D}$*) is the largest symmetric relation* $\mathcal{B}$ *such that* $P\,\mathcal{B}\,Q$ *implies*

– *if* $P \xrightarrow{C[-]} *_D P'$ *then* $Q \xrightarrow{C[-]} *_D Q'$ *and* $P'\,\mathcal{B}\,Q'$.

Note that the definition above requires that when P performs what is intuitively a τ-transition, i.e., when $P \xrightarrow{-}_D P'$, then a bisimilar process Q has to reply either with *one* or more τ-moves. Usually, the process Q might instead reply with *zero* or more τ-moves. This fact is discussed at the end of § 4.1.

3.3 Congruence Property

We now focus on adding constraints over $\mathbb{D}$ in order to make $\approx^D$ a congruence.

Definition 12. *Let* $\langle C_1[-], C_2[-]\rangle$ *be a decomposition pair for* $C'[C[P]] \rightsquigarrow P'$. *It is* universal *if* $\forall Q.\ Q \xrightarrow{C_1[-]}_D Q'' \Rightarrow C[Q] \xrightarrow{C'[-]}_D Q' \wedge Q' = C_2[Q'']$.

Consider the MAs process $P = \mathit{open}\, n.P_1$ and the context $C[-] = -|P_3$ and $C'[-] = -|n[P_2]$. The decomposition pair $\langle -|n[P_2], -|P_3\rangle$ of $C'[C[P]] \rightsquigarrow_M P_1|P_2|P_3$ is universal: for all Q s.t. $Q \xrightarrow{-|n[P_2]} Q''$ it holds $Q|P_3 \xrightarrow{-|n[P_2]} Q|P_3$.

Definition 13. *A context LTS* $\mathbb{D}$ *is* decomposable *if whenever* $C[P] \xrightarrow{C'[-]}_D P'$ *there exists a universal decomposition pair* $\langle C_1[-], C_2[-]\rangle$ *for* $C'[C[P]] \rightsquigarrow P'$.

The decomposition property introduced above is only required to prove the congruence property of the context bisimilarity (Proposition 2) and of the weak L-bisimilarity introduced in the next section (Proposition 3). The main result of our theory (namely, the labelled characterization of barbed bisimilarity stated in Theorem 1) just requires that the LTS is sound and complete.

Proposition 2. *If all contexts in* $\mathcal{C}$ *are reactive and* $\mathbb{D}$ *is decomposable, then* $\approx^D$ *is a congruence.*

Unfortunately, $\approx^D$ is often too fine-grained. Consider e.g. HoCore and the associated LTS H: $\approx^D$ is too discriminating for H. Indeed, as shown in § 2.2, the HoCore processes $a(x).\overline{a}x$ and $\mathbf{0}$ are asynchronously barbed congruent but they are obviously distinguished by $\approx^D$.

The right semantics is often represented by the barbed saturated one, whose flexibility allows for recasting a wide variety of observational, bisimulation-based equivalences. The main drawback of this semantics is the fact that in an equivalence proof all contexts must be tackled. For this reason, the next section introduces an alternative bisimilarity that efficiently characterizes the barbed saturated one, since it allows reasoning about it without considering all contexts.

4 Weak L-Bisimilarity

This section introduces weak L-bisimilarity, the weak counterpart of the bisimilarity proposed in [3]. It is parametric with respect to a set of contexts (also referred to as labels) L and it is a congruence, should L satisfy a closure property. More importantly, it can be used as an alternative proof technique of $\approx^{BS}$.

Definition 14 (Weak L-Bisimilarity). *Let $L \subseteq \mathcal{D}$ be a set of contexts.* Weak L-bisimilarity $\approx^L$ *(for $\mathbb{D}$) is the largest symmetric relation $\mathcal{B}$ s.t. $P\,\mathcal{B}\,Q$ implies*

$$\text{if } P \xrightarrow{C[-]}_D P' \text{ then } \begin{cases} Q \xrightarrow{C[-]} {}^*_D\, Q' \text{ and } P'\,\mathcal{B}\,Q', \text{ if } C[-] \in L; \\ C[Q] \rightsquigarrow^* Q' \text{ and } P'\,\mathcal{B}\,Q', \text{ otherwise.} \end{cases}$$

Note that $\approx^L$ generalizes $\approx^D$. Indeed, it is easy to prove that they coincide, when L is actually the whole set $\mathcal{D}$. In § 4.1 we show that for some L, weak L-bisimilarity also coincides with weak barbed saturated bisimilarity. We now show that $\approx^L$ is a congruence, under suitable conditions on L.

Definition 15. *A set of contexts $L \subseteq \mathcal{D}$ in a decomposable $\mathbb{D}$ is* closed under decomposition *if whenever $\langle C_1[-], C_2[-]\rangle$ is a universal decomposition pair for $C'[C[P]] \rightsquigarrow P'$ (see Definition 13) and $C'[-] \in L$ then $C_1[-] \in L$.*

Proposition 3. *Let L be a set of contexts in a decomposable $\mathbb{D}$. If all contexts in $\mathcal{C}$ are reactive and L is closed under decomposition, then $\approx^L$ is a congruence.*

4.1 Weak Barbed Saturated Bisimilarity via Weak L-Bisimilarity

Here we show that weak L-bisimilarity can characterize weak barbed saturated bisimilarity. It generalizes the correspondence between the strong version of the semantics, as it holds for Leifer and Milner's RSs [3]. This result is later used to prove that L-bisimilarity captures the right equivalences for MAs and HoCore.

The following definitions are needed to ensure that $\approx^L \subseteq \approx^{BS}$.

Definition 16 (Weak Contextual Barbs). *A barb o is a* weak contextual barb *if whenever $P \downarrow_o$ implies $Q \Downarrow_o$ then $\forall C[-]$, $C[P] \downarrow_o$ implies $C[Q] \Downarrow_o$.*

Definition 17. *Let L be a set of contexts. We say that L is* O-capturing *(for $\mathbb{D}$) if for each barb $o \in O$ there exists a context $C[-] \in L$ such that for each process P we have $P \downarrow_o$ if and only if $P \xrightarrow{C[-]}_D P'$.*

The next two definitions are instead needed to ensure that $\approx^{BS} \subseteq \approx^{L}$.

Definition 18. *Let $\mathcal{B}$ be a relation on processes. A predicate $\mathcal{P}(X,Y)$ on processes is* stable under $\mathcal{B}$ *(for $\mathbb{D}$) if for any two processes P, Q whenever $P\mathcal{B}Q$ and $\mathcal{P}(P,P')$ there exists Q' such that $\mathcal{P}(Q,Q')$ and $P'\mathcal{B}Q'$. A context $C[-]$ is* weakly stable under $\mathcal{B}$ *(for $\mathbb{D}$) if $\mathcal{P}(X,Y) = X \xrightarrow{C[-]}{}_D Y$ is stable under $\mathcal{B}$.*

In § 5 and 6, we show some contexts of MAs and HoCore that are weakly stable under $\approx^{BS}$. Note that the identity context $-$ is usually *not* stable under $\approx^{BS}$: for $P \approx^{BS} Q$ and $P \stackrel{-}{\rightarrow} P'$ (i.e., $P \rightsquigarrow P'$), it is not guaranteed that $Q \stackrel{-}{\rightarrow}$.

We may now state one of the main correspondence results for our theory, witnessing the usefulness of L-bisimilarity.

Theorem 1. *Let O be a set of weak contextual barbs and $L \subseteq \mathcal{D}$ a set of contexts in a sound and complete $\mathbb{D}$. If L is O-capturing and all its contexts are reactive and weakly stable under $\approx^{BS}$ (for $\mathbb{D}$), then $\approx^{BS} = \approx^{L}$.*

Theorem 1 requires that only the context of a subset L of $\mathcal{D}$ are reactive. However, sometimes it may be useful to consider only reactive contexts (like done implicitly in the graphical encodings of process calculi, see e.g [6]). This can be easily done, by observing that, as hinted above for MAs and HoCore, in most process calculi those contexts that are not reactive are also non-discriminating.

Remark 2. Note that weak L-bisimilarity coincides with weak context bisimilarity (Definition 11) when taking $L = \mathcal{D}$. Thus, as a consequence of the previous theorem, weak context bisimilarity coincides with weak barbed bisimilarity, whenever all the contexts in $\mathcal{D}$ are O-capturing and weakly stable under $\approx^{BS}$. However, this is usually not the case because, as discussed above, the identity context $-$ is not weakly stable. Indeed, in standard weak bisimilarities (e.g., the one for CCS), when one process perform one internal τ-transition (corresponding to $\stackrel{-}{\rightarrow}$), the equivalent process is not forced to perform any transition. By taking $L = \mathcal{D} \setminus \{-\}$, we capture exactly these standard weak bisimilarities (it suffices to observe that if $P \approx^{L} Q$ and $P \stackrel{-}{\rightarrow}_D P'$ then $Q \rightsquigarrow^* Q'$, since $- \notin L$) and via Theorem 1 we can check when they coincide with saturated barbed bisimilarity.

5 Labelled Characterization for MAs Weak Semantics

This section proposes a labelled characterization of the weak reduction barbed congruence for MAs by means of the weak L-bisimilarity over the LTS M.

First of all, we show that MAs fit in the theory of § 3. We consider the system theory $\mathbb{S}_M = \langle \mathcal{P}_M, \mathbb{C}_M, \cdot \rangle$, where $\mathbb{C}_M$ is the monoid $(\mathcal{C}_M, \circ, -)$, with $\mathcal{C}_M$ the set

of unary MAs contexts presented in § 2.1. The calculus can thus be seen as an RS $\mathcal{R}_M = \langle \mathbb{S}, \rightsquigarrow_M, O_M \rangle$, where O_M represents the set of MAs barbs.

It is easy to see that barbed saturated semantics for $\mathcal{R}_M$ coincides with the weak reduction barbed congruence for MAs.

Proposition 4. $\approx_M = \approx_M^{BS}$ *(for* $\mathcal{R}_M$*).*

The equivalence is thus a congruence with respect to all MAs contexts. However, in order to exploit the L-bisimilarity, we also have to show that M is a sound and complete context LTS. Instead of proving it directly, we consider a simpler RS, distilled in [2] by exploiting Leifer and Milner's theory. It differs from $\mathcal{R}_M$ only with respect to the monoid of contexts: $\mathcal{C}'_M$ contains the unary MAs contexts generated by the following grammar (for $R \in \mathcal{P}_M$)

$$C[-] ::= -,\ C[-]|R,\ (\nu n)C[-],\ n[C[-]].$$

We therefore consider the MAs system theory $\mathbb{S}'_M = \langle \mathcal{P}_M, \mathbb{C}'_M, \cdot \rangle$, where $\mathbb{C}'_M$ is the monoid $(\mathcal{C}'_M, \circ, -)$. The RS modelling MAs is now $\mathcal{R}'_M = \langle \mathbb{S}', \rightsquigarrow_M, O_M \rangle$ and M is a sound and complete context LTS based on $\mathcal{R}'_M$: its labels are "minimal" contexts, according to Leifer and Milner's terminology, and this fact ensures that both Property 1 (thanks to [11, Proposition 3]) and Property 2 (thanks to [11, Proposition 1]) of Definition 10 are satisfied.

Now, since the contexts of the shape $M.C[-]$ are non-discriminating, thanks to our Propositions 1 and 4, we can state the proposition below.

Proposition 5. $\approx_M = \approx_M^{BS}$ *(for* $\mathcal{R}'_M$*).*

As shown in § 4.1, we can characterize the weak barbed semantics on a set of weak contextual barbs O through a sound and complete context LTS and a set of reactive contexts L. As required by Theorem 1, L must be O-capturing and each $C[-] \in L$ must be weakly stable under weak barbed saturated bisimilarity.

Proposition 6. *Let* L_M *be the set of labels having the shape* $-|open\ n.T_1$*, for* n *an ambient name and* T_1 *an MAs process. Then,* L_M *is* O_M*-capturing. Moreover all the barbs in* O_M *are weak contextual.*

The proof of the proposition above occurs in [3, Proposition 8] ("L_M is O_M-capturing") and [4, Proposition 5] ("O_M are weak contextual"). It is easy to note that the contexts in L_M are reactive: it is now needed to prove that they are weakly stable under $\approx_M^{BS}$. To this end, we exploit the predicate in Fig. 6 that is equivalent to $\mathcal{P}(X, Y) = X \xrightarrow{C[-]} *_D Y$ and we show that it is stable under $\approx_M^{BS}$.

Lemma 1. *Let* $\mathcal{P}^{-|open\ n.T_1}(X, Y)$ *be the predicate on MAs processes shown in Fig. 6, for* n *ambient name and* T_1 *a process. Then, for any two processes* P *and* P' *we have* $\mathcal{P}^{-|open\ n.T_1}(P, P')$ *if and only if* $P \xrightarrow{-|open\ n.T_1} *_M P'$.

Proposition 7. *All labels in* L_M *are weakly stable under* $\approx_M^{BS}$.

From the previous results, the following theorem follows immediately.

Theorem 2. $\approx_M^{BS} = \approx^{L_M}$ *(for* $\mathcal{R}'_M$*).*

6 Labelled Characterizations for HoCore Semantics

As for MAs, we first show that HoCore fits in the theory of §3. We consider the HoCore system theory $\mathbb{S}_H = \langle \mathcal{P}_H, \mathbb{C}_H, \cdot \rangle$, where $\mathbb{C}_H$ is the monoid context $(\mathcal{C}_H, \circ, -)$, with $\mathcal{C}_H$ the set of unary HoCore contexts presented in § 2.2. The calculus can thus be seen as an RS $\mathcal{R}_H = \langle \mathbb{S}_H, \rightsquigarrow_H, O_H \rangle$, where O_H is the set of HoCore barbs defined in § 2.2.

It is easy to see that the weak barbed saturated semantics for $\mathcal{R}_H$ coincides with the weak asynchronous barbed congruence (Definition 2).

Proposition 8. $\sim_H^B = \approx_H^{BS}$ *(for* $\mathcal{R}_H$*).*

The LTS H (Fig. 5) is a sound and complete context LTS. Indeed, as for MAs, its labels are minimal contexts, according to Leifer and Milner's theory. For the sake of space we do not report the distillation procedure, but it is analogous to the one for CCS shown in [6]. However, as for MAs, the RS employed for distilling H slightly differs from $\mathcal{R}_H$, since there is no context of the shape $a(x).C[-]$.

We thus define $\mathcal{C}'_H$ as the set of contexts generated by $C[-] ::= -, C[-]|R$ (for $R \in \mathcal{P}_H$) and we consider the system theory $\mathbb{S}'_H = \langle \mathcal{P}_H, \mathbb{C}'_H, \cdot \rangle$, where $\mathbb{C}'_H$ is the monoid context $(\mathcal{C}'_H, \circ, -)$. Therefore, the RS modelling HoCore is now $\mathbb{R}'_H = \langle \mathbb{S}'_H, \rightsquigarrow_H, O_H \rangle$. Since contexts of the shape $a(x).C[-]$ are non-discriminating, Propositions 1 and 8 allow to state the following result.

Proposition 9. $\sim_H^B = \approx_H^{BS}$ *(for* $\mathcal{R}'_H$*).*

We now characterize $\sim_H^B$ via the LTS H (in Fig. 5) by choosing a set of contexts L_H and showing that the barbs in O_H are weak contextual and that L_H is O_H capturing and its contexts are reactive and weakly stable under $\approx_H^{BS}$.

In order to capture the barbs in O_H (defined as $P \downarrow \bar{a}$ iff $P \equiv \overline{a}P_1|P_2$) it suffices to take as set of contexts

$$L_H = \{- \mid a(x).S \text{ s.t. } a \in \mathcal{N},\ S \in \mathcal{P}_H\}.$$

Indeed, by definition of the LTS H, $P \xrightarrow{-|a(x).S}$ iff $P \downarrow \bar{a}$.

Proposition 10. *L_H is O_H-capturing and the barbs in O_H are weak contextual.*

It is easy to note that the contexts in L_H are reactive, so it only remains to prove that they are weakly stable under $\approx_H^{BS}$. To show this, we make use of the predicate in Fig. 7 and of the following lemma.

Lemma 2. *Let $\mathcal{P}^{-|a(x).S}(X,Y)$ be the predicate on* HoCore *processes shown in Fig. 7, for a name a and a process S. Then, for any two processes P and P' we have $\mathcal{P}^{-|a(x).S}(P,P')$ if and only if $P \xrightarrow{-|a(x).S} {}^*{}_H\ P'$.*

$\mathcal{P}^{-|open\ n.T_1}(X,Y) \quad \exists m \notin fn(X).\ X \mid open\ n.open\ m.T_1 \mid m[\mathbf{0}] \rightsquigarrow^* Y \wedge Y \not\Downarrow_m$

Fig. 6. Predicate for the labels $-|open\ n.T_1$

Proposition 11. *All labels in L_H are weak stable under $\approx_H^{BS}$.*

Theorem 3. $\approx_H^{BS} = \sim^{L_H}$ *(for $\mathbb{H}'_M$).*

It is interesting to remark that those contexts of the shape $-|\overline{a}Q$ (intuitively corresponding to the labels $a(x)$ in the ordinary LTS in Fig. 4) are not stable under $\approx_H^{BS}$. Indeed, $a(x).\overline{a}x \xrightarrow{-|\overline{a}Q}$ and $\mathbf{0} \not\xrightarrow{-|\overline{a}Q}$, even if $a(x).\overline{a}x \approx_H^{BS} \mathbf{0}$. The latter equivalence can now be formally proved by employing $\sim^{L_H}$: since the context $-|\overline{a}Q$ does not belong to L_H, whenever $a(x).\overline{a}x \xrightarrow{-|\overline{a}Q} \overline{a}Q$, the process $\mathbf{0}$ can reply with $\mathbf{0}|\overline{a}Q \rightsquigarrow^* \mathbf{0}|\overline{a}Q$ and clearly $\overline{a}Q \sim^{L_H} \mathbf{0}|\overline{a}Q$.

$\mathcal{P}^{-\|a(x).S}(X,Y)$	$\exists i \notin n(X)$ s.t. $X\|a(x).(S\|i(x).\mathbf{0}\|)\|\overline{i}\mathbf{0} \rightsquigarrow^* Y \not\Downarrow_{\overline{i}}$

Fig. 7. Predicates for the labels $-|a(x).S$

7 Conclusions and Future Work

Our paper investigates the notion of weak barbed semantics for RSs. More precisely, it proposes a general approach for the identification of suitable conditions under which weak barbed saturated bisimilarity can be characterized in terms of a behavioural equivalence over a suitably labelled transition system.

Weak barbed saturated bisimilarity generalizes the standard equivalences of calculi such as MAs and HoCore. Indeed, both case studies fall in our framework. In particular, for MAs we proved that our proposal captures its weak reduction barbed congruence, giving an alternative labelled characterization via weak L-semantics. For HoCore, after introducing the weak variant of the strong barbed semantics proposed in the original paper [10], we show that it can be captured via weak L-bisimilarity.

The two case studies also show an interesting problem of our approach that provides guidelines for future research. The LTSs M and H (in Figs. 2 and 5) are infinite, because the environment may provide infinitely many different S, S_1, S_2. Similar problems arise with different approaches (see e.g. [15,16]) and also for even more basic calculi. For instance, for the CCS shown in [6], to an ordinary input transition $\xrightarrow{a}$ correspond infinitely many contextual transitions $\xrightarrow{-|\bar{a}.S}$ and to an output $\xrightarrow{\bar{a}}$ correspond $\xrightarrow{-|a.S}$. However, in this simple case it suffices to take only the contexts where $S = \mathbf{0}$. The case of HoCore is slightly more complex but, similarly to CCS, one can check only few S, as in [10] with *normal bisimulation*. While for MAs, we are not aware of any "finite characterization" of its behavioural equivalence. This is also the case of many new-generation calculi featuring higher-order communication and hierarchical localities, such as the Kell [18]. Devising a general solution to this problem is a challenging task, that we would like to face with *open* reactive systems, in the style of [9].

References

1. Birkedal, L., Debois, S., Hildebrandt, T.T.: On the Construction of Sorted Reactive Systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 218–232. Springer, Heidelberg (2008)
2. Bonchi, F., Gadducci, F., Monreale, G.V.: Labelled transitions for mobile ambients (as synthesized via a graphical encoding). In: Hildebrandt, T., Gorla, D. (eds.) EXPRESS 2008. ENTCS, vol. 242(1), pp. 73–98. Elsevier, Amsterdam (2009)
3. Bonchi, F., Gadducci, F., Monreale, G.V.: On barbs and labels in reactive systems. In: Klin, B., Sobocinski, P. (eds.) SOS 2009. EPTCS, vol. 18, pp. 46–61 (2009)
4. Bonchi, F., Gadducci, F., Monreale, G.: Reactive Systems, Barbed Semantics, and the Mobile Ambients. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 272–287. Springer, Heidelberg (2009)
5. Bonchi, F., Montanari, U.: Symbolic semantics revisited. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 395–412. Springer, Heidelberg (2008)
6. Bonchi, F., Gadducci, F., König, B.: Synthesising CCS bisimulation using graph rewriting. Information and Computation 207(1), 14–40 (2009)
7. Cardelli, L., Gordon, A.: Mobile ambients. TCS 240(1), 177–213 (2000)
8. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science 16(6), 1133–1163 (2006)
9. Klin, B., Sassone, V., Sobociński, P.: Labels from Reductions: Towards a General Theory. In: Fiadeiro, J.L., Harman, N.A., Roggenbach, M., Rutten, J. (eds.) CALCO 2005. LNCS, vol. 3629, pp. 30–50. Springer, Heidelberg (2005)
10. Lanese, I., Pérez, J.A., Sangiorgi, D., Schmitt, A.: On the expressiveness and decidability of higher-order process calculi. Information and Computation 209(2), 198–226 (2011)
11. Leifer, J., Milner, R.: Deriving Bisimulation Congruences for Reactive Systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)
12. Merro, M., Zappa Nardelli, F.: Behavioral theory for mobile ambients. Journal of the ACM 52(6), 961–1023 (2005)
13. Milner, R., Sangiorgi, D.: Barbed Bisimulation. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 685–695. Springer, Heidelberg (1992)
14. Rathke, J., Sassone, V., Sobociński, P.: Semantic Barbs and Biorthogonality. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 302–316. Springer, Heidelberg (2007)
15. Rathke, J., Sobocinski, P.: Deconstructing behavioural theories of mobility. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, C.H.L. (eds.) IFP TCS 2008. IFIP, vol. 273, pp. 507–520. Springer, Heidelberg (2008)
16. Rathke, J., Sobocinski, P.: Deriving structural labelled transitions for mobile ambients. Information and Computation 208(10), 1221–1242 (2010)
17. Sassone, V., Sobociński, P.: Reactive systems over cospans. In: LICS 2005, pp. 311–320. IEEE Computer Society Press, Los Alamitos (2005)
18. Schmitt, A., Stefani, J.-B.: The Kell Calculus: A Family of Higher-Order Distributed Process Calcul. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 146–178. Springer, Heidelberg (2005)

Computation-by-Interaction with Effects

Ulrich Schöpp

Ludwig-Maximilians-Universität München, Munich, Germany
Ulrich.Schoepp@ifi.lmu.de

Abstract. A successful approach in the semantics of programming languages is to model programs by interaction dialogues. While dialogues are most often considered abstract mathematical objects, it has also been argued that they are useful for actual computation. A manual implementation of interaction dialogues can be complicated, however. To address this issue, we consider a general method for extending a given language with a metalanguage that supports the implementation of dialogues. This method is based on the construction by Dal Lago and the author of the programming language INTML, which applies interaction dialogues to sublinear space computation. We show that only few assumptions on the programming languages are needed to implement a useful INTML-like metalanguage. We identify a weak variant of the Enriched Effect Calculus (EEC) of Egger, Møgelberg & Simpson as a convenient setting for capturing the structure needed for the construction of the metalanguage. In particular, function types are not needed for the construction and iteration by means of a Conway operator is sufficient. By using EEC we show how computational effects can be accounted for in the implementation of interaction dialogues.

In game semantics and related areas of programming language semantics there is a long tradition of modelling programs by interaction dialogues. Programs are modelled as entities that may engage in a dialogue with their environment. The interpretation of a program explains what kinds of queries it can receive and how it may answer. Large programs are composed of smaller ones that interact with each other, so that the whole execution of a program may be considered an interaction process. The question/answer dialogues that make up such models tend to have very concrete nature, which has lead to interesting applications, for example in algorithmic game semantics.

The premise of this paper is that interaction dialogues are useful not only for interpreting programming languages, but also as an actual implementation method. There are many examples where dialogues have been used for the implementation of programs, e.g. [17,6,13]. Two recent examples provide the main motivation for the work reported here. First, Ghica introduces the Geometry of Synthesis [6] as a method of hardware synthesis. His approach is to construct a game model by implementing interaction dialogues by digital circuits and then to interpret a variant of Idealized Algol in the thus constructed game model. With this approach one can write a program in an high-level language (Idealized Algol) and by interpretation in the game model have it translated to a low-level language for digital circuits (Verilog). In this way, the implementation of dialogues is used as a method for hardware synthesis.

A similar example has been studied by Dal Lago and the author in the context of computation with sublinear space [13]. There the problem is how to write programs

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 305–321, 2011.

that operate on data too large to fit into memory. To access values that do not fit into memory one needs to query them piece by piece. Such computation can naturally be organised into question and answer dialogues. This observation has lead to the design of the programming language INTML for writing programs that can access and manipulate large data [13]. The approach is to start from a simple low-level language, whose programs can be evaluated using limited space, and use it to implement dialogue-based computation. The implementation of dialogues is considered as the construction of a game model, which is then used to interpret the language INTML. The result is that one can use the higher-order functional language INTML to implement dialogues in a simple low-level language that allows easy analysis of space usage.

These examples can be seen as game semantics turned around. Rather than interpreting a given programming language in a game model, one implements a game model in a given language. In the thus constructed model one can then interpret a new programming language. As the game model has been implemented in the original language, this new language may then be seen as a metalanguage for programming dialogues in the original language. The value of this approach is that even weak low-level languages suffice to construct rich game models that can interpret sophisticated metalanguages.

Having argued that it is useful to implement computation by question and answer dialogues, we turn to the problem of implementing dialogues in a given programming language. Motivated by the examples above, we focus in particular on weak languages that allow circuit synthesis or simple resource analysis or the like. From game semantics it is known that the concrete details of interaction dialogues can be complicated, so that in theoretical work it is standard practice to identify useful structure abstractly, e.g. products or function spaces, and to work with this abstract structure. Here we consider how a similar abstraction can be attained for the implementation of interaction dialogues.

We do this by reconsidering the ideas of INTML [13] in a more general context. The approach is to extend a given programming language with constructs that support the implementation of dialogues. These extensions are definitional in the sense that any program written with them could have been written without them, only perhaps in a more complicated way. The approach is thus to extend a given language with a language for metaprogramming. Previous evidence suggests that INTML captures useful language constructs for the implementation of interaction dialogues [14].

We show that very little structure is needed to carry out the construction of an INTML-like metalanguage and that computational effects can be allowed without affecting the metalanguage. Computational effects are interesting in this context, as can be seen from the examples above. Ghica's Geometry of Synthesis uses stateful circuits, while for sublinear space programming it is interesting to consider nondeterminism, perhaps in an effort to characterise the complexity class NLOGSPACE by a programming language. Beyond those examples, recent work on quantum λ-calculus [9] is based on computation by interaction with a quantum effect. Other effects, such as name generation, may be useful in the context of nominal game semantics [20]. There also seems to be a relation to Levy's Jump-with-Argument [16], which we intend to study in future work.

To give a rough idea of the metalanguage, assume given some programming language in which we want to implement the dialogues of a game model, e.g. PCF. The

metalanguage extends this language with a new class of types for interactive programming by dialogues. The new interaction types are formed by the grammar $X, Y ::= [A] \mid 1 \mid X \otimes Y \mid B \cdot X \multimap Y$, in which A and B range over types from the original language. Such an interaction type specifies an interface that tells which kinds of questions one may ask of programs of this type and which kinds of answer one may receive. The type $[A]$ specifies thunks that may be asked the single question 'please compute a value' and that may reply with any value of type A. The type $X \otimes Y$ contains pairs of values; it combines the interfaces of X and Y so that one may interact with either component of the pair as if one had two values of type X and Y side by side. The type $B \cdot X \multimap Y$ contains functions from X to Y. These functions are evaluated in an interactive manner, i.e. information about the function argument is obtained by sending questions according to its interface.

The type $B \cdot X \multimap Y$ imposes a linearity restriction on the use of its argument: it may be used B-many times, which means that there is one copy for each value of type B. The values of type B thus serve as addresses for the copies of X. The linearity constraint allows us to construct the metalanguage even in weak low-level languages that can only represent a limited number of addresses, e.g. languages with only finite types, as one would use for circuits. In strong languages like PCF, full copying can be allowed.

Some words are in order about why we study language extensions for metaprogramming as opposed to deriving completely new languages from the game models. While computation by dialogue is a useful mode of computation, it seems that not necessarily all computations should be done in this way. With a metalanguage one has the option of mixing computation by interaction with the usual computation of a given language. For example, it should be useful to have two function types, one that is evaluated using dialogues as in game semantics and the other one using standard call by value, say.

1 Weak Effect Calculus

Before describing the metalanguage in the next section, we define a weak effect calculus to capture the assumptions we make on the base programming language. An effect calculus suggests itself, as for the construction of the metalanguage we need possibly non-terminating loops and so must account for the effect of non-termination at least.

We define the Weak Effect Calculus (WEC), a weak variant of the Enriched Effect Calculus (EEC), which was introduced by Egger, Møgelberg & Simpson [4] as a type theory for studying computational effects. The Enriched Effect Calculus develops Moggi's computational metalanguage [19] and can also be understood as a reformulation and extension of Levy's Call-by-Push-Value (CBPV) [16]. The choice of (a variant of) EEC as a basis for the construction of the metalanguage is motivated mainly by its clean separation of values and computations as well as the presence of copower types, which are particularly useful.

The Weak Effect Calculus (WEC) is obtained by taking the fragment of EEC without function types and products of computation types and adding sum types for values and a Conway operator for iteration:

$$\textbf{Value types} \quad A, B ::= \alpha \mid 0 \mid A + B \mid 1 \mid A \times B \mid \underline{A}$$
$$\textbf{Computation types} \quad \underline{A}, \underline{B} ::= \underline{\alpha} \mid \underline{0} \mid \underline{A} \oplus \underline{B} \mid A \cdot \underline{B} \mid !A$$

It may be useful to think of value types simply as sets and computation types as sets with an additional element $\perp$ intended to represent non-termination.

For value types we choose the usual sum and product types. For computation types we take sums ($\underline{0}$ and $\underline{A} \oplus \underline{B}$), copowers $A \cdot \underline{B}$ and computations $!A$. The type $A \cdot \underline{B}$ can be thought of as a type of pairs of a value of type A and a computation of type $\underline{B}$. The computation type $!A$ consists of computations that when executed may return a value of type A. It plays the role of TA in Moggi's computational λ-calculus. Further types could be added without affecting the results in this paper.

Like the Enriched Effect Calculus, WEC has two kinds of judgements, $\Gamma \mid - \vdash f : A$ and $\Gamma \mid x : \underline{B} \vdash g : \underline{C}$. Both contain a context Γ that assigns *value types* to variables. In addition there is a *stoup* that may either be empty or that may consist of a single variable declaration of computation type. The first judgement declares f to be a value of value type A. The second judgement declares g to be a computation. The term g therein may be thought of as an evaluation context whose hole is identified by x. An operational intuition is that the evaluation of g starts with the complete evaluation of x and then continues to evaluate g. In the above-mentioned interpretation of computation types as sets with an element $\perp$ for non-termination, g appears as a strict function.

The terms and typing rules for WEC are given in Fig. 1. Therein, Δ ranges over stoups and may be either empty $-$ or a variable declaration $x : \underline{B}$. The rules are subject to the condition that only judgements of one of the two forms above can be derived. For example, in the elimination rule for 0 the stoup Δ can only be empty if A is a computation type.

In addition to the usual terms for the various types from EEC, WEC also contains a term $(\mathsf{let}\ x = f\ \mathsf{loop}\ g)$ for iteration. The operational intuition is that first f is evaluated, its result is bound to x and then g is evaluated. If the result is $\mathsf{inl}(h)$, then h is the result of $(\mathsf{let}\ x = f\ \mathsf{loop}\ g)$. If the result is $\mathsf{inr}(f')$, then the computation continues as $(\mathsf{let}\ x = f'\ \mathsf{loop}\ g)$. In this way, the term $(\mathsf{let}\ x = f\ \mathsf{loop}\ g)$ represents a looping computation that may, in particular, fail to terminate.

This form of iteration will be enough to support the construction of a metalanguage in the next section. For example for the space usage analysis results in [13] it is essential that such a form of iteration suffices and full recursion is not needed.

In WEC the meaning of terms is explained by an equational theory. Except for the term $(\mathsf{let}\ x = f\ \mathsf{loop}\ g)$ the equations of WEC are just as for EEC, see [4,5]. The equations for the term $(\mathsf{let}\ x = f\ \mathsf{loop}\ g)$ are those of a uniform Conway operator [21]. A Conway operator is a mapping $(-)^\dagger$ that takes a morphism $\underline{A} \to \underline{B} \oplus \underline{A}$ in some category, where $\oplus$ denotes the coproduct, to a morphism $\underline{A} \to \underline{B}$ in the same category, subject to a number of equations[1]. The term $(\mathsf{let}\ x = f\ \mathsf{loop}\ g)$ is syntax for such a Conway operator, for given a term $\Gamma \mid x : \underline{A} \vdash g : \underline{B} \oplus \underline{A}$ one can form $\Gamma \mid y : \underline{A} \vdash \mathsf{let}\ x = y\ \mathsf{loop}\ g : \underline{B}$.

There are six equations for such a Conway operator: the fixpoint property, naturality, dinaturality, diagonal and uniformity. The fixpoint property is expressed by:

[1] In loc. cit. Conway operators studied in the dual setting, taking $\underline{B} \times \underline{A} \to \underline{A}$ to $\underline{B} \to \underline{A}$.

$$\frac{}{\Gamma, x\colon A \mid - \vdash x\colon A} \qquad \frac{}{\Gamma \mid x\colon \underline{A} \vdash x\colon \underline{A}} \qquad \frac{}{\Gamma \mid - \vdash *\colon 1}$$

$$\frac{\Gamma \mid - \vdash f\colon A \qquad \Gamma \mid - \vdash g\colon B}{\Gamma \mid - \vdash \langle f, g\rangle\colon A \times B} \qquad \frac{\Gamma \mid - \vdash f\colon A \times B}{\Gamma \mid - \vdash \mathsf{fst}(f)\colon A} \qquad \frac{\Gamma \mid - \vdash f\colon A \times B}{\Gamma \mid - \vdash \mathsf{snd}(f)\colon B}$$

$$\frac{\Gamma \mid - \vdash f\colon 0}{\Gamma \mid \Delta \vdash \mathsf{image}(f)\colon A} \qquad \frac{\Gamma \mid - \vdash f\colon A}{\Gamma \mid - \vdash \mathsf{inl}(f)\colon A + B} \qquad \frac{\Gamma \mid - \vdash f\colon B}{\Gamma \mid - \vdash \mathsf{inr}(f)\colon A + B}$$

$$\frac{\Gamma \mid - \vdash f\colon A + B \qquad \Gamma, x\colon A \mid \Delta \vdash g\colon C \qquad \Gamma, y\colon B \mid \Delta \vdash h\colon C}{\Gamma \mid \Delta \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h\colon C}$$

$$\frac{\Gamma \mid \Delta \vdash f\colon \underline{0}}{\Gamma \mid \Delta \vdash \mathsf{image}(f)\colon \underline{A}} \qquad \frac{\Gamma \mid \Delta \vdash f\colon \underline{A}}{\Gamma \mid \Delta \vdash \mathsf{inl}(f)\colon \underline{A} \oplus \underline{B}} \qquad \frac{\Gamma \mid \Delta \vdash f\colon \underline{B}}{\Gamma \mid \Delta \vdash \mathsf{inr}(f)\colon \underline{A} \oplus \underline{B}}$$

$$\frac{\Gamma \mid \Delta \vdash f\colon \underline{A} \oplus \underline{B} \qquad \Gamma \mid x\colon \underline{A} \vdash g\colon \underline{C} \qquad \Gamma \mid y\colon \underline{B} \vdash h\colon \underline{C}}{\Gamma \mid \Delta \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow g \mid \mathsf{inr}(y) \Rightarrow h\colon \underline{C}}$$

$$\frac{\Gamma \mid - \vdash f\colon A \qquad \Gamma \mid \Delta \vdash g\colon \underline{B}}{\Gamma \mid \Delta \vdash f \cdot g\colon A \cdot \underline{B}} \qquad \frac{\Gamma \mid \Delta \vdash f\colon A \cdot \underline{B} \qquad \Gamma, x\colon A \mid y\colon \underline{B} \vdash g\colon \underline{C}}{\Gamma \mid \Delta \vdash \mathsf{let}\ x \cdot y = f\ \mathsf{in}\ g\colon \underline{C}}$$

$$\frac{\Gamma \mid - \vdash f\colon A}{\Gamma \mid - \vdash\ !f\colon\ !A} \qquad \frac{\Gamma \mid \Delta \vdash f\colon\ !A \qquad \Gamma, x\colon A \mid - \vdash g\colon \underline{B}}{\Gamma \mid \Delta \vdash \mathsf{let}\ !x = f\ \mathsf{in}\ g\colon \underline{B}}$$

$$\frac{\Gamma \mid \Delta \vdash f\colon \underline{A} \qquad \Gamma \mid x\colon \underline{A} \vdash g\colon \underline{B} \oplus \underline{A}}{\Gamma \mid \Delta \vdash \mathsf{let}\ x = f\ \mathsf{loop}\ g\colon \underline{B}}$$

Fig. 1. Typing Rules of the Weak Effect Calculus

$$\frac{\Gamma \mid \Delta \vdash f\colon \underline{A} \qquad \Gamma \mid x\colon \underline{A} \vdash g\colon \underline{B} \oplus \underline{A}}{\Gamma \mid \Delta \vdash (\mathsf{let}\ x = f\ \mathsf{loop}\ g)\ =\ \mathsf{case}\ g[f/x]\ \mathsf{of}\ \begin{cases}\mathsf{inl}(x) \Rightarrow x\\ \mathsf{inr}(y) \Rightarrow \mathsf{let}\ x = y\ \mathsf{loop}\ g\end{cases} : \underline{A}}$$

The other four equations are just syntactic formulations of the corresponding equations in [21]. We omit them, as they are needed only to construct a uniform trace [7]. We could have added a uniform trace to WEC directly, but Conway operators appear more natural in the syntax, for example for giving an operational semantics.

1.1 Implementing Interaction

Suppose now we want to implement in WEC a way of computation by interaction, where the interface of an entity is given by a pair $(\underline{X}^-, \underline{X}^+)$ of computation types. Think of $\underline{X}^-$ as the type of questions that may be asked of the entity and $\underline{X}^+$ as the type of possible answers.

A term of type $\Gamma \mid z\colon \underline{Y}^- \oplus \underline{X}^+ \vdash f\colon \underline{Y}^+ \oplus \underline{X}^-$ then implements a strategy of answering questions for an entity with interface $Y = (\underline{Y}^-, \underline{Y}^+)$ when given the ability to ask questions of an entity with interface $X = (\underline{X}^-, \underline{X}^+)$. For, suppose we have a term $\Gamma \mid x\colon \underline{X}^- \vdash e\colon \underline{X}^+$ that answers questions for X. Then we can define a term $\Gamma \mid x\colon \underline{Y}^- \vdash g\colon \underline{Y}^+$ that answers questions for Y. Concretely, we can define g to be the term $\mathsf{let}\ z = \mathsf{inl}(y)\ \mathsf{loop}\ \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(y) \Rightarrow \mathsf{inl}(y) \mid \mathsf{inr}(x) \Rightarrow \mathsf{inr}(\mathsf{inl}(e))$. Of course,

this term is not very easy to read, nor are such terms easy to write. The metalanguage in the next section provides language constructs for writing such programs.

2 A Metalanguage for Interactive Computation

We now introduce INTML[WEC], a metalanguage for implementing interactive computation in WEC. Formally we do this by introducing a new class of interaction types as well as a new typing judgement for interactive computations. We will then show that these extensions are in fact definitional, i.e. can be implemented in the original calculus. In this way, the new constructs can be seen as constructs for metaprogramming WEC.

As outlined in the Introduction, we add four kinds of interaction types:

$$\textbf{Interaction type} \quad X, Y ::= [A] \mid 1 \mid X \otimes Y \mid A \cdot X \multimap Y$$

Each interaction type X represents a pair $(\underline{X}^-, \underline{X}^+)$ of computation types that specifies the interface the value of type X:

$$1^- = \underline{0} \quad [A]^- = !1 \quad (X \otimes Y)^- = \underline{X}^- \oplus \underline{Y}^- \quad (A \cdot X \multimap Y)^+ = A \cdot \underline{X}^- \oplus \underline{Y}^+$$
$$1^+ = \underline{0} \quad [A]^+ = !A \quad (X \otimes Y)^+ = \underline{X}^+ \oplus \underline{Y}^+ \quad (A \cdot X \multimap Y)^- = A \cdot \underline{X}^+ \oplus \underline{Y}^-$$

Interaction sequents have the form $\Gamma \mid x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n \vdash^{\mathsf{i}} t : Y$. The context Γ maps variables to value types, as before. In the second part of the context, each variable x_i appears with multiplicity A_i, which is a value type. This means that x_i represents A_i-many copies of X_i, one for each value of A_i. The term t explains how to answer questions for Y given the ability to ask questions of the various copies of X_i.

The typing rules are given in Figs. 2, 3 and 4. In the rules we write $A \cdot \Phi$ for the context obtained by replacing each declaration $x : B \cdot X$ in Φ with $x : (A \times B) \cdot X$. In rule (STRUCT) we use a relation $A \leq B$ that informally expresses that B-many copies of X are more than A-many copies. Since A and B are value types, we formalise this by requiring there to exist a section-retraction pair between A and B.

$$\textsc{Var}\ \frac{}{\Gamma \mid x : 1 \cdot X \vdash^{\mathsf{i}} x : X} \qquad \textsc{Struct}\ \frac{\Gamma \mid \Phi, x : A \cdot X \vdash^{\mathsf{i}} t : Y}{\Gamma \mid \Phi, x : B \cdot X \vdash^{\mathsf{i}} t : Y}\ A \leq B$$

$$\textsc{Weak}\ \frac{\Gamma \mid \Phi \vdash^{\mathsf{i}} t : Y}{\Gamma \mid \Phi, x : 1 \cdot X \vdash^{\mathsf{i}} t : Y} \qquad \textsc{Exch}\ \frac{\Gamma \mid \Phi, x : A \cdot X, y : B \cdot Y, \Psi \vdash^{\mathsf{i}} t : Z}{\Gamma \mid \Phi, y : B \cdot Y, x : A \cdot X, \Psi \vdash^{\mathsf{i}} t : Z}$$

$$\textsc{Copy}\ \frac{\Gamma \mid \Phi \vdash^{\mathsf{i}} s : X \qquad \Gamma \mid \Psi, x : A \cdot X, y : B \cdot X \vdash^{\mathsf{i}} t : Z}{\Gamma \mid \Psi, (A + B) \cdot \Phi \vdash^{\mathsf{i}} \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t : Z}$$

Fig. 2. Typing Rules of INTML[WEC]: Structural Rules

The terms of INTML[WEC] represent strategies for computation by interaction. For most part the typing rules define a simply-typed λ-calculus with restricted copying. This λ-calculus interacts with the base language WEC by the rules ($[\,]$I) and ($[\,]$E). The operational intuition for the latter is: when asked to compute an answer in $[B]$, first ask s for its value. Upon receipt of an answer, which must be a value of type A, bind

$$[]\mathrm{I}\ \frac{\Gamma \mid - \vdash f : !A}{\Gamma \mid - \vdash^{\mathsf{i}} [f] : [A]} \qquad []\mathrm{E}\ \frac{\Gamma \mid \Phi \vdash^{\mathsf{i}} s : [A] \qquad \Gamma, x : A \mid \Psi \vdash^{\mathsf{i}} t : [B]}{\Gamma \mid \Phi, A \cdot \Psi \vdash^{\mathsf{i}} \mathsf{let}\ [x] = s\ \mathsf{in}\ t : [B]}\ 1 \leq A$$

$$\otimes\mathrm{I}\ \frac{\Gamma \mid \Phi \vdash^{\mathsf{i}} s : X \qquad \Gamma \mid \Psi \vdash^{\mathsf{i}} t : Y}{\Gamma \mid \Phi, \Psi \vdash^{\mathsf{i}} \langle s, t \rangle : X \otimes Y} \qquad \otimes\mathrm{E}\ \frac{\begin{array}{c}\Gamma \mid \Phi \vdash^{\mathsf{i}} s : X \otimes Y \\ \Gamma \mid \Psi, x : A \cdot X, y : A \cdot Y \vdash^{\mathsf{i}} t : Z\end{array}}{\Gamma \mid \Psi, A \cdot \Phi \vdash^{\mathsf{i}} \mathsf{let}\ \langle x, y \rangle = s\ \mathsf{in}\ t : Z}$$

$$\multimap\mathrm{I}\ \frac{\Gamma \mid \Phi, x : A \cdot X \vdash^{\mathsf{i}} t : Y}{\Gamma \mid \Phi \vdash^{\mathsf{i}} \lambda x.t : A \cdot X \multimap Y} \qquad \multimap\mathrm{E}\ \frac{\begin{array}{c}\Gamma \mid \Psi \vdash^{\mathsf{i}} s : X \\ \Gamma \mid \Phi \vdash^{\mathsf{i}} t : A \cdot X \multimap Y\end{array}}{\Gamma \mid \Phi, A \cdot \Psi \vdash^{\mathsf{i}} t\, s : Y}\ 1 \leq A$$

$$1\mathrm{I}\ \frac{}{\Gamma \mid - \vdash^{\mathsf{i}} * : 1} \qquad \textsc{Direct}\ \frac{\Gamma \mid x : \underline{X}^{-} \vdash f : \underline{X}^{+}}{\Gamma \mid - \vdash^{\mathsf{i}} \mathsf{direct}(x.f) : X}$$

Fig. 3. Typing Rules of INTML[WEC]: Introductions, Eliminations and Direct Definition

$$0\mathrm{E}^{\mathsf{i}}\ \frac{\Gamma \mid - \vdash f : 0}{\Gamma \mid \Phi \vdash^{\mathsf{i}} \mathsf{image}(f) : X} \qquad []\mathrm{E}^{\mathsf{c}}\ \frac{\Gamma \mid - \vdash^{\mathsf{i}} s : [A] \qquad \Gamma, x : A \mid - \vdash f : \underline{B}}{\Gamma \mid - \vdash \mathsf{let}\ [x] = s\ \mathsf{in}\ f : \underline{B}}$$

$$+\mathrm{E}^{\mathsf{i}}\ \frac{\Gamma \mid - \vdash f : A + B \qquad \Gamma, x : A \mid \Phi \vdash^{\mathsf{i}} s : X \qquad \Gamma, y : B \mid \Phi \vdash^{\mathsf{i}} t : X}{\Gamma \mid \Phi \vdash^{\mathsf{i}} \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t : X}$$

Fig. 4. Typing Rules of INTML[WEC]: Cross-Eliminations

the result to x and query t. The answer, a value of type B, is then passed on to answer the initial request.

In this way the computational effects from WEC become available in INTML[WEC]. The term let $[x] = [f]$ in t represents a computation that first executes the computation $f : !A$ thus performing its effects and then continues with the execution of t.

That INTML[WEC] allows only restricted copying is important in order to be able to include weak low-level languages as base languages. With further assumptions about the base language, it is possible to allow full copying. Indeed, if there exists a value type G with $G \times G \leq G$ and $G + G \leq G$ and $A \leq G$ for any other value type A, then INTML[WEC] allows full copying. For instance, one could take for G a type of natural numbers that can encode the values of all value types. This approach is familiar from Geometry of Interaction Situations [1]. Another option, naturally suggested by the requirements $G \times G \leq G$ and $G + G \leq G$, is to use for G a type of trees. Such a type of trees is used by Mackie [17] in an interactive implementation of PCF.

However, fine-grained control of copying is an important feature of INTML[WEC]. This form of copying works even when there is no type G as above. In particular, the metalanguage is well-behaved even for very basic base languages, for example ones having only finite types. For applications such as programming with sublinear space or circuit synthesis it is essential to be able to account for such weak languages. Furthermore, while the multiplicity annotations $A \cdot (-)$ in the types complicate the type system, previous experience with INTML suggest that the type system nevertheless remains manageable [14], as type inference is possible.

$$
\begin{aligned}
&(1\text{-}\beta)\ \Gamma \mid \Phi \vdash^{\mathsf{i}} s = *\colon 1 \text{ if } \Gamma \mid \Phi \vdash^{\mathsf{i}} s\colon 1\\
&(\otimes\text{-}\beta)\ \Gamma \mid \Phi, \Psi \vdash^{\mathsf{i}} (\mathsf{let}\ \langle x_1, x_2\rangle = \langle t_1, t_2\rangle\ \mathsf{in}\ x_k) = t_k\colon X_k \text{ for } k = 1, 2\\
&\qquad \text{if } \Gamma \mid \Phi \vdash^{\mathsf{i}} t_1\colon X_1 \text{ and } \Gamma \mid \Psi \vdash^{\mathsf{i}} t_2\colon X_2\\
&(\otimes\text{-}\eta)\ \Gamma \mid \Phi, A\cdot\Psi \vdash^{\mathsf{i}} (\mathsf{let}\ \langle x, y\rangle = s\ \mathsf{in}\ t[\langle x, y\rangle/z]) = t[s/z]\colon Z\\
&\qquad \text{if } \Gamma \mid \Phi \vdash^{\mathsf{i}} s\colon X \otimes Y \text{ and } \Gamma \mid \Psi, x\colon A\cdot X, y\colon A\cdot Y \vdash^{\mathsf{i}} t\colon Z\\
&(\multimap\text{-}\beta)\ \Gamma \mid \Phi, A\cdot\Psi \vdash^{\mathsf{i}} (\lambda x.\, s)\, t = s[t/x]\colon Y\\
&\qquad \text{if } \Gamma \mid \Phi, x\colon A\cdot X \vdash^{\mathsf{i}} s\colon Y \text{ and } \Gamma \mid \Psi \vdash^{\mathsf{i}} t\colon X\\
&(\multimap\text{-}\eta)\ \Gamma \mid \Phi \vdash^{\mathsf{i}} (\lambda x.\, t\, x) = t\colon A\cdot X \multimap Y \text{ if } \Gamma \mid \Phi \vdash^{\mathsf{i}} t\colon A\cdot X \multimap Y \text{ and } x \notin \Phi.\\
&([\,]\text{-}\beta)\ \Gamma \mid A\cdot\Phi \vdash^{\mathsf{i}} (\mathsf{let}\ [x] = [!v]\ \mathsf{in}\ t) = t[v/x]\colon [B],\\
&\qquad \Gamma \mid - \vdash (\mathsf{let}\ [x] = [!v]\ \mathsf{in}\ f) = f[v/x]\colon \underline{C}\\
&\qquad \text{if } \Gamma \mid - \vdash v\colon A \text{ and } \Gamma, x\colon A \mid \Phi \vdash^{\mathsf{i}} t\colon [B] \text{ and } \Gamma, x\colon A \mid - \vdash f\colon \underline{C}\\
&(0^{\mathsf{i}}\text{-}\beta)\ \Gamma \mid \Phi \vdash^{\mathsf{i}} \mathsf{image}(f) = s[f/x]\colon X \text{ if } \Gamma \mid - \vdash f\colon 0 \text{ and } \Gamma, x\colon 0 \mid \Phi \vdash^{\mathsf{i}} s\colon X\\
&(+^{\mathsf{i}}\text{-}\beta)\ \Gamma \mid \Phi \vdash^{\mathsf{i}} (\mathsf{case}\ \mathsf{inl}(f)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t) = s[f/x]\colon X,\\
&\qquad \Gamma \mid \Phi \vdash^{\mathsf{i}} (\mathsf{case}\ \mathsf{inr}(f)\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow s \mid \mathsf{inr}(y) \Rightarrow t) = t[f/y]\colon X\\
&\qquad \text{if } \Gamma \mid - \vdash f\colon A \text{ and } \Gamma, x\colon A \mid \Phi \vdash^{\mathsf{i}} s\colon X \text{ and } \Gamma, y\colon B \mid \Phi \vdash^{\mathsf{i}} t\colon X\\
&(\textsc{Cp})\ \Gamma \mid \Psi) \vdash^{\mathsf{i}} \mathsf{copy}\ s[\boldsymbol{t}/\boldsymbol{x}]\ \mathsf{as}\ y, y'\ \mathsf{in}\ u = \mathsf{copy}\ \boldsymbol{t}\ \mathsf{as}\ \boldsymbol{x}, \boldsymbol{x}'\ \mathsf{in}\ u[s/y, s[\boldsymbol{x}'/\boldsymbol{x}]/y']\colon Z\\
&\qquad \text{if } \left\{\Gamma \mid \Phi_i \vdash^{\mathsf{i}} t_i\colon X_i\right\}_{i=1}^{n} \text{ and } \Gamma \mid x_1\colon A_1\cdot X_1, \ldots, x_n\colon A_n\cdot X_n \vdash^{\mathsf{i}} s\colon Y \text{ and}\\
&\qquad \Gamma \mid \Psi, y\colon B\cdot Y, y'\colon C\cdot Y \vdash^{\mathsf{i}} u\colon Z \text{ and } \Psi = (B + C)\cdot(A_1\cdot\Phi_1, \ldots, A_n\cdot\Phi_n)
\end{aligned}
$$

Fig. 5. Equations of INTML[WEC]

INTML[WEC] improves over INTML not only in generality but also in terms of its equational theory. While in INTML only equations between closed terms are given and justified by a semantic interpretation, here we include equations between open terms, as one would expect from a well-behaved type theory, see Fig. 5.

A simply-typed λ-calculus alone is not very expressive as a programming language, of course. The intention in INTML[WEC] is that further constructs can be added by the programmer by direct implementation of combinators. Rule (DIRECT) allows a programmer to define combinators directly by implementing a strategy for it in the base language. Game semantics is a rich source of such strategies. For example, it is possible to implement combinators for loops, for control operators, or for locally scoped state:

$$
\begin{aligned}
\texttt{loop} &: \alpha\cdot(\gamma\cdot[\alpha] \multimap [\alpha+\beta]) \multimap [\alpha] \multimap [\beta]\\
\texttt{callcc} &: \big(\gamma\cdot([\alpha] \multimap [\beta]) \multimap [\alpha]\big) \multimap [\alpha]\\
\texttt{newvar} &: \alpha\cdot\big(\delta\cdot\big((\gamma\cdot[\alpha] \multimap [1]) \otimes [\alpha]\big) \multimap [\beta]\big) \multimap [\beta]
\end{aligned}
$$

These and other combinators can be implemented by direct definition, e.g. $\texttt{loop} = \mathsf{direct}(x.f)$, for suitable f. The definitions of `loop` and `callcc` are described in [13], In essence, `callcc` implements a game semantic strategy described also by Laird [15]. The combinator `newvar` represents a memory cell of type α. It is intended to be used as $\texttt{newvar}(\lambda\langle\mathsf{write}, \mathsf{read}\rangle.\, t)$. In t the memory cell can be read by means of $\mathsf{let}\ [x] = \mathsf{read}\ \mathsf{in}\ s$ and written with value v by $\mathsf{let}\ [*] = (\mathsf{write}\ [v])\ \mathsf{in}\ s$.

Other than congruences there are no equations for direct in INTML[WEC], as this term allows one to implement arbitrary strategies. Equations for specific direct-terms such as `loop`, `callcc` or `newvar` have to be considered on a case-by-case basis.

The above combinators are good examples why it is useful to have value types as bounds for copying, as opposed to natural numbers, say. In `newvar`, for example, the

content of the memory cell is encoded in the number of the argument. Hence, there are α-many copies of the argument, as indicated by the type.

Example. To give a simple concrete example of the use of the metalanguage, we consider the Kierstead terms, which are often used to illustrate the need for justification pointers in Hyland-Ong games [10] for modelling higher-order λ-calculus. With explicit copying, these terms can be given the following types, where α can be any nonempty value type:

$$t_1 = \lambda f.\,\mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\, f_2\ (\lambda y.\, y)) :\ (1+\alpha)\cdot(\alpha\cdot(1\cdot X \multimap X)\multimap X)\multimap X$$
$$t_2 = \lambda f.\,\mathsf{copy}\ f\ \mathsf{as}\ f_1, f_2\ \mathsf{in}\ f_1\ (\lambda x.\, f_2\ (\lambda y.\, x)) :\ (1+\alpha)\cdot(\alpha\cdot(\alpha\cdot X \multimap X)\multimap X)\multimap X$$

What is encoded by the justification pointers in Hyland-Ong games is here, as in Abramsky-Jagadeesan-Malacaria games, encoded in the copy of the function argument.

That these two terms do indeed implement different strategies can be shown by constructing an argument for which t_1 and t_2 give different results. To do this, define the value type $\mathsf{bool} = 1 + 1$ with abbreviations $\mathtt{tt}$ and $\mathtt{ff}$ for its two elements. It is easy to define in WEC a term $x\colon \mathsf{bool},\ y\colon \mathsf{bool} \mid - \vdash \mathsf{nor}(x,y)\colon !\mathsf{bool}$ that returns $\mathtt{tt}$ when both x and y are $\mathtt{ff}$ and $\mathtt{ff}$ otherwise. If we then define the function f of type $(1 + \mathsf{bool})\cdot(\beta\cdot[\mathsf{bool}]\multimap[\mathsf{bool}])\multimap[\mathsf{bool}]$ to be $\lambda g.\,\mathsf{copy}\ g\ \mathsf{as}\ g_1, g_2\ \mathsf{in}\ \mathsf{let}\ [x_1] = g_1\ [\mathtt{ff}]\ \mathsf{in}\ \mathsf{let}\ [x_2] = g_2\ [\mathtt{tt}]\ \mathsf{in}\ [\mathsf{nor}(x_1,x_2)]$, then we have $t_1\ f = [\mathtt{tt}]$ and $t_2\ f = [\mathtt{ff}]$.

3 Models of the Weak Effect Calculus

Having defined INTML[WEC] as a metalanguage for interaction, our goal is now to show that the Weak Effect Calculus from Sec. 1 can implement this language. We shall do so by a semantic argument, showing that from any model of WEC we can construct a model for INTML[WEC]. A translation from INTML[WEC] to WEC follows by applying this result to a term model of WEC.

We briefly define the semantic structure needed to model WEC. These definitions are a straightforward adaptation of those for EEC in [4]. The notion of a model of WEC uses basic enriched category theory [12].

Values are modelled in a small category $\mathbb{V}$ in a standard way. To account for the indexing of computations over value contexts, computations are modelled in a $\widehat{\mathbb{V}}$-enriched category $\mathbb{C}$, where $\widehat{\mathbb{V}} = \mathbf{Set}^{\mathbb{V}^{\mathrm{op}}}$ is the category of presheaves over $\mathbb{V}$ [18]. The $\widehat{\mathbb{V}}$-enrichment of $\mathbb{C}$ accounts for the indexing over value contexts, as the hom-object $\mathbb{C}(\underline{A},\underline{B})$ is now a presheaf, i.e. a functor from $\mathbb{V}^{\mathrm{op}}$ to $\mathbf{Set}$. For each value type Γ we have a set $\mathbb{C}(\underline{A},\underline{B})(\Gamma)$. The terms of type $\Gamma \mid x\colon \underline{A} \vdash f\colon \underline{B}$ appear as elements of this set.

The category $\mathbb{V}$ itself can also be seen as a $\widehat{\mathbb{V}}$-enriched category by $\mathbb{V}(A,B) = (\mathbf{y}B)^{(\mathbf{y}A)}$, where $\mathbf{y}A = \mathbb{V}(-,A)$ is the Yoneda embedding. Using the Yoneda lemma one can see that this enrichment can be spelled out as $\mathbb{V}(A,B)(\Gamma) \simeq \mathbb{V}(\Gamma\times A, B)$.

A model for WEC then consists of a $\widehat{\mathbb{V}}$-enriched adjunction $F \dashv U\colon \mathbb{V}\to\mathbb{C}$ with the following structure:

1. In $\mathbb{V}$: finite products and finite coproducts which distributive over products; this induces $\widehat{\mathbb{V}}$-enriched finite products and $\widehat{\mathbb{V}}$-enriched finite coproducts in $\mathbb{V}$.

2. In $\mathbb{C}$: $\widehat{\mathbb{V}}$-enriched finite coproducts and copowers indexed by representables. The latter means that for each object A in $\mathbb{V}$ and each object $\underline{B}$ in $\mathbb{C}$ there is an object $A \cdot \underline{B}$ in $\mathbb{C}$ and an isomorphism $\mathbb{C}(\underline{B}, \underline{C})^{\mathbf{y}A} \simeq \mathbb{C}(A \cdot \underline{B}, \underline{C})$ that is $\widehat{\mathbb{V}}$-natural in $\underline{C}$. As in the syntax of WEC, we write $\oplus$ and $\underline{0}$ for binary coproducts and initial object.
3. In $\mathbb{C}^{\mathrm{op}}$: a uniform parametrised Conway operator in the sense that there is a map $(-)^\dagger \colon \mathbb{C}(\underline{A}, \underline{B} \oplus \underline{A}) \to \mathbb{C}(\underline{A}, \underline{B})$ in $\widehat{\mathbb{V}}$ that when considered as a map of type $\mathbb{C}^{\mathrm{op}}(\underline{B} \oplus \underline{A}, \underline{A}) \to \mathbb{C}^{\mathrm{op}}(\underline{B}, \underline{A})$ satisfies the equations of a uniform Conway operator [21]. Note that $\oplus$ is a product in $\mathbb{C}^{\mathrm{op}}$.
4. We require this structure to be such that the canonical maps $\mathbb{C}(X, Y)^{\mathbf{y}0} \to 1$ and $\mathbb{C}(X, Y)^{\mathbf{y}(A+B)} \to \mathbb{C}(X, Y)^{\mathbf{y}A} \times \mathbb{C}(X, Y)^{\mathbf{y}B}$ are isomorphisms. This requirement is used to model the elimination of 0 and $A + B$ over computations.

Value and computation types are interpreted in this structure as objects of $\mathbb{V}$ and $\mathbb{C}$ respectively. If a computation type is used as a value type, then this is modelled by an application of the functor U. In the other direction, the type $!A$ is interpreted by FA. A value sequent $\Gamma \mid - \vdash f \colon A$ is interpreted as an element of $\mathbb{V}(1, A)(\Gamma)$. A computation sequent $\Gamma \mid x \colon \underline{A} \vdash g \colon \underline{B}$ appears in the model as an element of $\mathbb{C}(\underline{A}, \underline{B})(\Gamma)$. While value sequents $\Gamma \mid - \vdash f \colon \underline{B}$ defining a term of computation type are interpreted as elements of $\mathbb{V}(1, U\underline{B})(\Gamma)$, by the adjunction $F \dashv U$ their interpretation is in one-to-one correspondence with $\mathbb{C}(F1, \underline{B})(\Gamma)$, so that they may also be seen a computations.

A simple example of a model can be obtained by letting $\mathbb{V}$ be the category of finite sets and $\mathbb{C}$ be the $\widehat{\mathbb{V}}$-category of finite pointed sets and strict functions, i.e. an object of $\mathbb{C}$ is a finite set $\underline{A}$ with a distinguished element $\bot$ and $\mathbb{C}(\underline{A}, \underline{B})(\Gamma)$ consists of all functions $f \colon \Gamma \times \underline{A} \to \underline{B}$ that satisfy $f(\gamma, \bot) = \bot$ for any $\gamma \in \Gamma$.

A second example is a term model. The objects of $\mathbb{V}$ and $\mathbb{C}$ are the value types and computation types respectively. The morphisms from A to B in $\mathbb{V}$ are terms $x \colon A \mid - \vdash f \colon B$, identified up to equality. Likewise, $\mathbb{C}(\underline{A}, \underline{B})(C)$ consists of terms $x \colon C \mid y \colon \underline{A} \vdash g \colon \underline{B}$ identified up to provable equality.

As the notation suggests, we use the copower to interpret copying in INTML[WEC]. We have found that the copower identifies just the right structure for this purpose. For any object A define $\mathbb{C}^A$ to be the $\widehat{\mathbb{V}}$-category with the same objects as $\mathbb{C}$ and with hom-objects $\mathbb{C}^A(\underline{B}, \underline{C}) = \mathbb{C}(\underline{B}, \underline{C})^{\mathbf{y}A}$. Just as $\mathbb{C}$ models sequents of the form $\Gamma \mid y \colon \underline{B} \vdash f \colon \underline{C}$, $\mathbb{C}^A$ models sequents of the form $\Gamma, x \colon A \mid y \colon \underline{B} \vdash f \colon \underline{C}$. There is a canonical $\widehat{\mathbb{V}}$-functor $W_A \colon \mathbb{C} \to \mathbb{C}^A$, that amounts to weakening. Then, the copower extends to a $\widehat{\mathbb{V}}$-functor $A \cdot (-) \colon \mathbb{C}^A \to \mathbb{C}$ that is left adjoint to W_A (in a $\widehat{\mathbb{V}}$-enriched sense). As a left adjoint, $A \cdot (-)$ preserves sums. Moreover, we have a canonical isomorphism $A \cdot FB \simeq F(A \times B)$, as $A \times (-)$ is a copower in $\mathbb{V}$ and copowers, being a particular form of colimit, are preserved by the left adjoint F.

Lemma 1. *In $\mathbb{C}$ there are the isomorphisms $1 \cdot \underline{C} \simeq \underline{C}$ and $(A \times B) \cdot \underline{C} \simeq A \cdot (B \cdot \underline{C})$ and $0 \cdot \underline{C} \simeq 0$ and $(A + B) \cdot \underline{C} \simeq A \cdot \underline{C} + B \cdot \underline{C}$, which are all natural in A, B and $\underline{C}$.*

To establish the last two isomorphisms we use the assumption in point 4 of the definition of a model above.

3.1 Trace and Int-Construction

The first step in constructing a model of INTML[WEC] from a model of WEC is to apply the Int-construction [11] to the category of computations. This construction is well-known in the context of game semantics; it has been used by Abramsky and Jagadeesan to model AJM-games [2].

Lemma 2. *The $\widehat{\mathbb{V}}$-category $\mathbb{C}$ has a uniform trace with respect to coproducts as monoidal structure, in the sense that there is a map $Tr_{\underline{B},\underline{C},\underline{D}}\colon \mathbb{C}(\underline{B}\oplus\underline{C},\underline{D}\oplus\underline{C}) \to \mathbb{C}(\underline{B},\underline{D})$ in $\widehat{\mathbb{V}}$ that satisfies the usual equations for a uniform trace [7].*

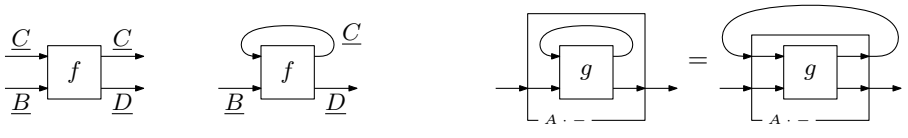
Such a uniform trace can be constructed from a uniform Conway operator, as has been shown (in a dual setting) by Hasegawa [7].

Lemma 3. *If $\mathbb{C}$ has a uniform trace with respect to coproducts then so does $\mathbb{C}^A$ and both $\widehat{\mathbb{V}}$-functors $W_A\colon \mathbb{C} \to \mathbb{C}^A$ and $A\cdot(-)\colon \mathbb{C}^A \to \mathbb{C}$ preserve the trace.*

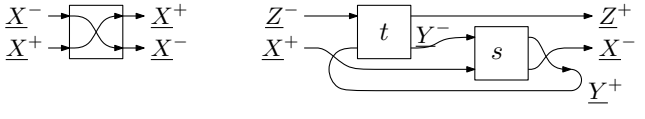
For the proof that $A\cdot(-)$ preserves the trace we need the assumption of uniformity.

It is useful to use a graphical notation for working with the traced monoidal category $\mathbb{C}$. We denote an element $f \in \mathbb{C}(\underline{B}\oplus\underline{C},\underline{D}\oplus\underline{C})(\Gamma)$ as in the box on left below and use similar standard notation for the traced monoidal structure. For example, the result of applying the trace to f is shown next to f below. However, note that these diagrams are now used to work in $\widehat{\mathbb{V}}$, so that care is needed to verify, e.g., naturality conditions.

For the copower functor we use a box-notation as shown in the equation on the right below. In that equation $g \in \mathbb{C}^A(\underline{B}\oplus\underline{C},\underline{D}\oplus\underline{E})(\Gamma) \simeq \mathbb{C}(\underline{B}\oplus\underline{C},\underline{D}\oplus\underline{E})(\Gamma\times A)$. The equation expresses that $A\cdot(-)$ preserves the trace. The box-notation is justified, as $A\cdot(-)$ is a monoidal functor with respect to coproducts.



The Int-construction then defines a $\widehat{\mathbb{V}}$-category $\mathrm{Int}(\mathbb{C})$ as follows. An object X is a pair $(\underline{X}^-,\underline{X}^+)$ of $\mathbb{C}$-objects. The hom-objects for $\mathrm{Int}(\mathbb{C})$ are defined by $\mathrm{Int}(\mathbb{C})(X,Y) = \mathbb{C}(\underline{Y}^-\oplus\underline{X}^+,\underline{Y}^+\oplus\underline{X}^-)$. The definition of the identity $1 \to \mathrm{Int}(\mathbb{C})(X,X)$ and the composition $\mathrm{Int}(\mathbb{C})(Y,Z)\times\mathrm{Int}(\mathbb{C})(X,Y) \to \mathrm{Int}(\mathbb{C})(X,Z)$ are best understood when given as graphical diagrams in $\mathbb{C}$:



Lemma 4. *$Int(\mathbb{C})$ is a $\widehat{\mathbb{V}}$-category with a monoidal closed structure $(I,\otimes,\multimap)$ with $I = (\underline{0},\underline{0})$, $X\otimes Y = (\underline{X}^- + \underline{Y}^-, \underline{X}^+ + \underline{Y}^+)$ and $X \multimap Y = (\underline{X}^+ + \underline{Y}^-, \underline{X}^- + \underline{Y}^+)$.*

The structure in this lemma is well-known, see e.g. [8].

To model copying in INTML[WEC] we use in addition a functor $X^{\otimes A}$ that informally captures an A-fold tensor $X \otimes \cdots \otimes X$. To define it formally, define a category $\mathbb{V}_{\mathrm{sr}}$ of all section-retraction-pairs in $\mathbb{V}$, i.e. a morphism from A to B is a pair $\langle s, r\rangle \in \mathbb{V}(A,B) \times \mathbb{V}(B,A)$ with $r \circ s = id$. Note that $\mathbb{V}_{\mathrm{sr}}$ is again canonically $\widehat{\mathbb{V}}$-enriched.

We define the object $X^{\otimes A}$ to be $(A \cdot \underline{X}^-, A \cdot \underline{X}^+)$. This definition can be extended to a functor in each argument, i.e. to $(-)^{\otimes A}\colon \mathrm{Int}(\mathbb{C}) \to \mathrm{Int}(\mathbb{C})$ and $X^{\otimes(-)}\colon (\mathbb{V}_{\mathrm{sr}})^{\mathrm{op}} \to \mathrm{Int}(\mathbb{C})$. That $\mathbb{V}_{\mathrm{sr}}$ consists of section-retraction pairs ensures functoriality of this definition. We note that these two functors do *not* combine to a bifunctor $(-)^{\otimes(-)}$.

As $X^{\otimes A}$ amounts to repeated multiplication, there are isomorphisms that correspond to well-known rules of high-school arithmetic:

$$\begin{array}{llll} X^{\otimes 1} \simeq X & X^{\otimes(A\times B)} \simeq \left(X^{\otimes A}\right)^{\otimes B} & I^{\otimes A} \simeq I & \\ X^{\otimes 0} \simeq I & X^{\otimes(A+B)} \simeq X^{\otimes A} \otimes X^{\otimes B} & (X \otimes Y)^{\otimes A} \simeq X^{\otimes A} \otimes Y^{\otimes A} & \end{array} \tag{1}$$

These isomorphisms follow from Lemma 1. They are natural in A, B, X and Y.

4 Interpreting the Metalanguage in WEC

Starting from $\mathrm{Int}(\mathbb{C})$, we now build a model that can interpret INTML[WEC]. While the terms of INTML[WEC] can already be interpreted in $\mathrm{Int}(\mathbb{C})$, this interpretation validates only equations between closed terms. For example, given $\Sigma \mid \Phi \vdash^{\mathsf{i}} s\colon X$ and $\Sigma \mid \Psi \vdash^{\mathsf{i}} t\colon Y$ it does not have to be the case that $\Sigma \mid \Phi, \Psi \vdash^{\mathsf{i}} \mathsf{let}\ \langle x, y\rangle = \langle s, t\rangle\ \mathsf{in}\ x\colon X$ and $\Sigma \mid \Phi, \Psi \vdash^{\mathsf{i}} s\colon X$ receive the same denotation. This is because we may start a dialogue with these terms by sending messages to the variables of t, i.e. those in Ψ. In the first term these messages are processed by t, while in the second term they are just discarded. We would like to discount such differences, as they appear only if one answers questions that have never been asked.

Another reason why $\mathrm{Int}(\mathbb{C})$ does not justify open equations is that in the interpretation of rule (STRUCT) an arbitrary section-retraction pair between A and B may be chosen and different such choices can be observed in $\mathrm{Int}(\mathbb{C})$. However, different choices cannot affect the final result of computations, so that we would like to consider them implementation details that should not be taken into account when considering program equality.

In order to explain in which sense we consider the behaviour of programs equal, we now take a quotient of the model with respect to a form of logical relations. This quotient is similar to the ones taken in AJM-games [2], but with WEC with unspecified effects as a basis, we cannot use an equivalence relation on traces of values as in [2].

The following definition is to be understood internally in the presheaf topos $\widehat{\mathbb{V}}$. In it we denote by $|\mathbb{C}|$ the discrete category with the same objects as $\mathbb{C}$.

Definition 1. *A* Kripke partial equivalence relation *with arity* $\mathcal{I}\colon |\mathbb{C}| \to \mathit{Int}(\mathbb{C})$ *over an object* X *in* $\mathit{Int}(\mathbb{C})$ *is a family of partial equivalence relations*

$$\left(R(\underline{A}) \subseteq \mathit{Int}(\mathbb{C})(\mathcal{I}\underline{A}, X) \times \mathit{Int}(\mathbb{C})(\mathcal{I}\underline{A}, X)\right)_{\underline{A} \in |\mathbb{C}|} .$$

In the following we use such Kripke partial equivalence relations with respect to the arity $\mathcal{I}$ defined by $\mathcal{I}(\underline{A}) = (\underline{A}, \underline{0})$. Notice that an element of $\mathrm{Int}(\mathbb{C})(\mathcal{I}\underline{A}, X)$ corresponds to a map in $\mathbb{C}(\underline{X}^-, \underline{X}^+ \oplus \underline{A})$. We use such maps, as opposed to just $\mathbb{C}(\underline{X}^-, \underline{X}^+)$ in order to avoid making the quotient too strong. Without the presence of $\underline{A}$ a morphism modelling `callcc` would be ruled out, for example.

We fix some notation. First note that, for any $\underline{A}$, there is a canonical morphism $\Delta_{\underline{A}} \colon \mathcal{I}(\underline{A}) \to \mathcal{I}(\underline{A}) \otimes \mathcal{I}(\underline{A})$. Similarly, there is a canonical morphism $\Delta_{\underline{A}}^{\otimes B} \colon \mathcal{I}(\underline{A}) \to \mathcal{I}(\underline{A})^{\otimes B}$. Given $e \colon \mathcal{I}\underline{A} \to X$ we write short $e^{\otimes B}$ for $e^{\otimes B} \circ \Delta_{\underline{A}}^{\otimes B} \colon \mathcal{I}\underline{A} \to X^{\otimes B}$.

Given a small model of WEC, we construct a $\widehat{\mathbb{V}}$-category $\mathbb{I}$, which can model INTML[WEC]. The construction works by restricting and quotienting $\mathrm{Int}(\mathbb{C})$ in the style of models based on partial equivalence relations.

The objects of $\mathbb{I}$ are pairs (X, R_X) of an *underlying object* X in $\mathrm{Int}(\mathbb{C})$ and a Kripke partial equivalence relation over X. The $\widehat{\mathbb{V}}$-object $\mathbb{I}(X, Y)$ of morphisms from X to Y consists of the quotient of $\mathrm{Int}(\mathbb{C})(X, Y)$ under the partial equivalence relation $\sim$ defined as follows in the internal logic of $\widehat{\mathbb{V}}$:

$$f \sim g \iff \forall \underline{A}.\, \forall (e, e') \in R_X(\underline{A}).\, (f \circ e, g \circ e') \in R_Y(\underline{A})$$

We define in $\mathbb{I}$ objects 1, $[B]$, $X \otimes Y$, $X \multimap Y$ and $X^{\otimes A}$ as follows. The underlying object of 1 is I and the underlying objects of the others are the $\mathrm{Int}(\mathbb{C})$-object of the same name. The relations are defined by:

$$\begin{aligned}
R_1(\underline{A}) &= \top \\
R_{[B]}(\underline{A}) &= \{(e, e) \mid e \in \mathrm{Int}(\mathbb{C})(\mathcal{I}\underline{A}, [B])\} \\
R_{X \otimes Y}(\underline{A} \oplus \underline{B}) &= \{(e_1 \otimes e_2, e_1' \otimes e_2') \mid (e_1, e_1') \in R_X(\underline{A}) \wedge (e_2, e_2') \in R_Y(\underline{B})\} \\
R_{X \multimap Y}(\underline{A}) &= \{(f, f') \mid \forall \underline{B}.\, \forall (e, e') \in R_X(\underline{B}).\, (\varepsilon \circ \langle f, e \rangle, \varepsilon \circ \langle f', e' \rangle) \in R_Y(\underline{A} \oplus \underline{B})\} \\
R_{X^{\otimes B}}(\underline{A}) &= \{({e_1}^{\otimes B}, {e_2}^{\otimes B}) \mid (e_1, e_2) \in R_X(\underline{A})\}
\end{aligned}$$

We use these constructions for the interpretation of INTML[WEC] in $\mathbb{I}$. The types 1, $[A]$ and $X \otimes Y$ are interpreted by the corresponding objects. The function type $A \cdot X \multimap Y$ is interpreted by the object $(X^{\otimes A}) \multimap Y$. A term $\Gamma \mid x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n \vdash^{\mathsf{i}} t \colon Y$ is interpreted as an element of $\mathbb{I}({X_1}^{\otimes A_1} \otimes \cdots \otimes {X_n}^{\otimes A_n}, Y)(\Gamma)$.

Lemma 5. *$[-]$ is a $\widehat{\mathbb{V}}$-functor from $Kl(T)$ to $\mathbb{I}$, where $Kl(T)$ is the $\widehat{\mathbb{V}}$-category with same objects as $\mathbb{V}$ and with $Kl(T)(A, B) = \mathbb{C}(FA, FB)$.*

Lemma 6. *$(1, \otimes, \multimap)$ defines a symmetric monoidal closed structure in $\mathbb{I}$ whose unit 1 is terminal.*

The definition of $X^{\otimes A}$ in $\mathbb{I}$ is such that it informally represents A-many copies of the *same* element of X. As reordering such tuples of several copies of the same element has no effect, we may replace $\mathbb{V}_{\mathrm{sr}}$ by a preorder $\leq$ on $\mathbb{V}$-objects. It is defined such that $A \leq B$ holds if and only if there is a morphism from A to B in $\mathbb{V}_{\mathrm{sr}}$. We write $\mathbb{V}_{\leq}$ for the category arising from this preorder.

Lemma 7. *The definition of $X^{\otimes B}$ extends to a $\mathbb{V}$-functor $(-)^{\otimes(-)} \colon \mathbb{I} \times \mathbb{V}_{\leq}^{\mathrm{op}} \to \mathbb{I}$ and there are morphisms that are natural in A, B, X and Y.*

$$X^{\otimes 1} \xrightarrow{\simeq} X \qquad X^{\otimes(A\times B)} \xrightarrow{\simeq} \left(X^{\otimes A}\right)^{\otimes B} \qquad 1^{\otimes A} \xrightarrow{\simeq} 1$$
$$X^{\otimes 0} \xrightarrow{\simeq} 1 \qquad X^{\otimes(A+B)} \to X^{\otimes A} \otimes X^{\otimes B} \qquad (X \otimes Y)^{\otimes A} \xrightarrow{\simeq} X^{\otimes A} \otimes Y^{\otimes A}$$

The next lemma follows immediately from the definition of the morphisms of $\mathbb{I}$ as equivalence classes under $\sim$.

Lemma 8. *The functor $U \colon \mathbb{I} \to \widehat{\mathbb{V}}$ that maps an object X to the $\mathbb{V}$-set of equivalence classes of $\bigcup_{\underline{A}} R_X(\underline{A})$ and a morphism $[f]_\sim \colon X \to Y$ to the function $[e] \mapsto [f \circ e]$ is faithful and maps the monoidal structure $(1, \otimes, \multimap)$ to $(1, \times, \Rightarrow)$.*

Note in particular that U sends X and $X^{\otimes A}$ to isomorphic objects. The isomorphisms from Lemma 7 are all mapped to the identity; $X^{\otimes(A+B)} \to X^{\otimes A} \otimes X^{\otimes B}$ is mapped to the diagonal. When reasoning about the identity of maps in $\mathbb{I}$ we therefore do not need to consider the explicit duplication by means of $X^{\otimes A}$.

The next two lemmas capture the structure needed to interpret rules ([ ]I), ([ ]E) and the β-equation $(\mathsf{let}\ [y] = [!x]\ \mathsf{in}\ t) = t[x/y]$.

Lemma 9. *There is an isomorphism $\varphi \colon \mathbb{I}(1, [B]) \xrightarrow{\simeq} Kl(T)(1, B)$ that is natural in B.*

Proof. Note that $\mathbb{I}(1, [B])$ is isomorphic to $\mathrm{Int}(\mathbb{C})(1, [B])$, which by definition is isomorphic to $\mathbb{C}(\underline{0} \oplus F1, FB \oplus \underline{0})$. By definition of $Kl(T)(1, B)$ the result follows. □

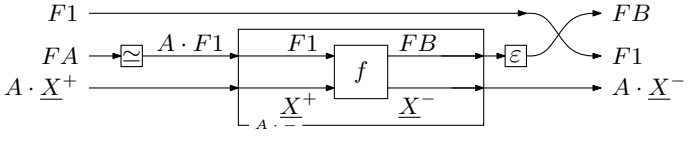
Lemma 10. *There exists a map $\psi \colon \mathbb{I}(X, [B])^{\mathbf{y}A} \to \mathbb{I}(X^{\otimes A} \otimes [A], [B])$ that is natural in X and that any square of the following form commutes.*

$$\begin{array}{ccc} \mathbb{I}(X, [B])^{\mathbf{y}A} & \xrightarrow{\langle i \circ \psi, r\rangle} & \mathbb{I}(X^{\otimes A} \otimes [A], [B])^{\mathbf{y}A} \times \mathbb{I}(1, [A])^{\mathbf{y}A} \\ {\scriptstyle \langle id, l\rangle}\downarrow & & \downarrow \\ \mathbb{I}(X, [B])^{\mathbf{y}A} \times \mathbb{I}(X^{\otimes A}, X)^{\mathbf{y}A} & \longrightarrow & \mathbb{I}(X^{\otimes A}, [B])^{\mathbf{y}A} \end{array}$$

Therein r and l are the canonical composites $r \colon 1 \to Kl(1, A)^{\mathbf{y}A} \to \mathbb{I}(1, [A])^{\mathbf{y}A}$ and $l \colon 1 \to \mathbb{V}(1, A)^{\mathbf{y}A} = \mathbb{V}_{\mathrm{sr}}(1, A)^{\mathbf{y}A} \to \mathbb{I}(X^{\otimes A}, X^{\otimes 1})^{\mathbf{y}A} \simeq \mathbb{I}(X^{\otimes A}, X)^{\mathbf{y}A}$ and the unlabelled maps are the canonical maps of their type.

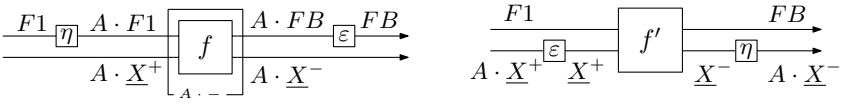
We spell out the proof since it gives a good example of how to work with $\mathbb{I}$ and because it illustrates that the copower fits in very well with the string diagrams.

Proof. Since morphisms in $\mathbb{I}$ are equivalence classes, we define ψ on representatives and observe that the definition does not depend on the choice of representative. Given $f \in \mathrm{Int}(\mathbb{C})(X, [B])^{\mathbf{y}A}(\Gamma)$, define $\psi(f)$ to be the equivalence class of the following morphism in $\mathrm{Int}(\mathbb{C})(X^{\otimes A} \otimes [A], [B])(\Gamma)$.

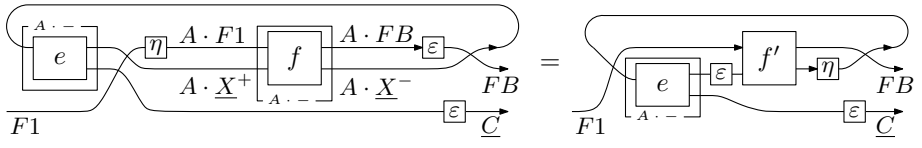


Here η and ε denote the unit and counit of the adjunction $A \cdot (-) \dashv W_A$. The box labelled with $\simeq$ denotes the canonical isomorphism arising as F preserves copowers.

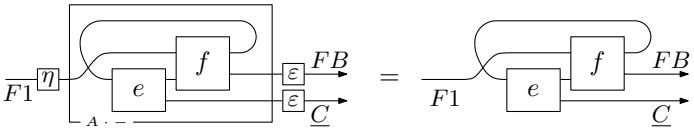
To show that the square in the lemma commutes, let $f \sim f'$ be two representatives of an equivalence class in $\mathbb{I}(X, [B])^{\mathbf{y}A}(\Gamma)$. The two composites in the square then give the following two maps in $\mathbb{I}(X^{\otimes A}, [B])^{\mathbf{y}A}(\Gamma)$.



We have to show that they are $\sim$-related, which amounts to showing that $(e, e') \in R'_X(\underline{C})$ implies the following equality:



We can simplify the left morphism left by joining the two $A \cdot (-)$-boxes, using functoriality and naturality of the trace. We obtain the morphism on the left below. By noting that the adjunction $A \cdot (-) \dashv W_A$ gives us $W_A(\varepsilon) \circ W_A(A \cdot h) \circ \eta$ for any h by naturality of η and the triangular identity, we obtain the equality below (in which W_A is implicit).



The right morphism in the equation that we have to show can similarly be simplified. We obtain the same result with f' instead of f. The result then follows from $f \sim f'$. □

For the interpretation of $+$-cross-elimination we use the following lemma.

Lemma 11. *The canonical* $\mathbb{I}(X, Y)^{\mathbf{y}(A+B)} \to \mathbb{I}(X, Y)^{\mathbf{y}A} \times \mathbb{I}(X, Y)^{\mathbf{y}B}$ *is isomorphic.*

The interpretation of INTML[WEC] is now defined such that the types and terms of WEC are interpreted in $\mathbb{V}$ and $\mathbb{C}$, as in [4]. The terms of the metalanguage are interpreted in $\mathbb{I}$.

Theorem 1 (Soundness). *The construction of* $\mathbb{I}$ *as quotient of Int*$(\mathbb{C})$ *yields a model of* INTML[WEC]. *That is,* INTML[WEC] *can be soundly interpreted in* $\mathbb{I}$*, given that rule (*DIRECT*) is restricted to define only morphisms in* $\mathbb{I}$.

The proof uses standard lemmas for substitution and for showing that two derivations of the same sequent have the same interpretation. A translation of INTML[WEC] to WEC is obtained by starting the construction of $\mathbb{I}$ with the term model of WEC.

5 Conclusion

We have shown that a simple variant of the enriched effect calculus provides enough structure to support the construction of a metalanguage INTML[WEC] for interaction. This improves on the previous construction of INTML in a number of ways. We show that any computational effect that justifies the equations of WEC can be added to the base language. We justify equations between open terms in INTML[WEC], not just closed equations as in [13], by means of a quotient in $\mathrm{Int}(\mathbb{C})$. In contrast to the construction in [13], we do not need to consider an operational semantics of the base language and make the construction using an equational theory only. Finally, we show that the structure of INTML can be accounted for in an enriched setting by a relatively simple model construction. In particular the copower type from EEC turns out to be the right structure for modelling bounded copying in the metalanguage.

Acknowledgments. Rasmus Møgelberg first noticed a similarity of EEC and INTML and suggested to study their relationship. INTML was developed with Ugo Dal Lago [13]. I thank the anonymous referees for their interesting comments and suggestions.

References

1. Abramsky, S., Haghverdi, E., Scott, P.J.: Geometry of interaction and linear combinatory algebras. Mathematical Structures in Computer Science 12(5), 625–665 (2002)
2. Abramsky, S., Jagadeesan, R.: New foundations for the geometry of interaction. Inf. Comput. 111(1), 53–119 (1994)
3. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)
4. Egger, J., Møgelberg, R.E., Simpson, A.: Enriching an Effect Calculus with Linear Types. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 240–254. Springer, Heidelberg (2009)
5. Egger, J., Møgelberg, R.E., Simpson, A.: Linearly-Used Continuations in the Enriched Effect Calculus. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 18–32. Springer, Heidelberg (2010)
6. Ghica, D.R.: Geometry of synthesis: a structured approach to VLSI design. In: POPL, pp. 363–375. ACM (2007)
7. Hasegawa, M.: Models of Sharing Graphs: A Categorical Semantics of let and letrec. Distinguished Dissertation Series. Springer, Heidelberg (1999)
8. Hasegawa, M.: On traced monoidal closed categories. Mathematical Structures in Computer Science 19(2), 217–244 (2009)
9. Hasuo, I., Hoshino, N.: Semantics of Higher-Order Quantum Computation via Geometry of Interaction. In: LICS (2011)
10. Hyland, J.M.E., Luke Ong, C.-H.: On full abstraction for PCF: I, II, and III. Inf. Comput. 163(2), 285–408 (2000)
11. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Math. Proc. Cambridge Philos. Soc. 119(3), 447–468 (1996)
12. Kelly, G.M.: Basic Concepts of Enriched Category Theory. Lecture Notes in Mathematics, vol. 64. Cambridge University Press (1982)
13. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 205–225. Springer, Heidelberg (2010)

14. Dal Lago, U., Schöpp, U.: Type inference for sublinear space functional programming. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 376–391. Springer, Heidelberg (2010)
15. Laird, J.: Full abstraction for functional languages with control. In: LICS, pp. 58–67 (1997)
16. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis. Semantics Structures in Computation, vol. 2. Springer, Heidelberg (2004)
17. Mackie, I.: The geometry of interaction machine. In: POPL, pp. 198–208 (1995)
18. MacLane, S., Moerdijk, I.: Sheaves in geometry and logic: A first introduction to topos theory. Springer, Heidelberg (1994)
19. Moggi, E.: Computational lambda-calculus and monads. In: LICS, pp. 14–23. IEEE Computer Society (1989)
20. Murawski, A.S., Tzevelekos, N.: Algorithmic Nominal Game Semantics. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 419–438. Springer, Heidelberg (2011)
21. Simpson, A., Plotkin, G.: Complete axioms for categorical fixed-point operators. In: LICS, pp. 30–41 (2000)

Towards a Certified Petri Net Model-Checker

Lukasz Fronc and Franck Pommereau

IBISC, University of Évry, Tour Évry 2
523 place des terrasses de l'Agora, 91000 Évry, France
{fronc,pommereau}@ibisc.univ-evry.fr

Abstract. *Petri nets* are widely used in the domain of automated verification through *model-checking*. In this approach, a Petri Net model of the system of interest is produced and its reachable states are computed, searching for erroneous executions. *Model compilation* can accelerate this analysis by generating code to explore the reachable states. This avoids the use of a fixed exploration tool involving an "interpretation" of the Petri net structure. In this paper, we show how to compile Petri nets targeting the *LLVM language* (a high-level assembly language) and formally prove the *correctness* of the produced code. To this aim, we define a *structural operational semantics* for the fragment of LLVM we use.

Keywords: explicit model-checking, model compilation, LLVM, SOS.

1 Introduction

Verification through *model-checking* [1] consists in defining a formal model of the system to be analysed and then using automated tools to check whether the expected properties are met or not. We consider here more particularly the widely adopted setting in which models are expressed using *coloured Petri nets* [9] and there states are explored using *explicit model-checking* that enumerates them all (contrasting with symbolic model-checking that works with sets of states). *Model compilation* is one of the numerous techniques to speedup explicit model-checking, it relies on generating source code (then compiled into machine code) to produce a high-performance implementation of the state space exploration primitives. For instance, this approach is successfully used in the well known SPIN tool [7], or in Helena coloured Petri net model-checker [14,3].

In this paper, we propose a way to prove the correctness of such an approach. More precisely, we focus on the produced code and prove that the object computed by its execution is an actual representation of the state space of the compiled model. We consider the *Low-Level Virtual Machine* (*LLVM*) language as our target language for compilation, which reconciles two otherwise contradictory objectives: on the one hand, this is a typed language with reasonably high-level operations allowing to express algorithms quite naturally; on the other hand, it is a low-level language that can be equipped with a formal semantics allowing to formally prove the programs correctness. To do so, we define a structural operational semantics of the fragment of LLVM we need and use it to establish the properties of the programs generated by our compiler.

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 322–336, 2011.

To the best of our knowledge, this is the first attempt to provide a formal semantics for LLVM. Moreover, if model-checkers are widely used tools, there exists surprisingly few attempts to prove them at the implementation level [19], contrasting with the domain of proof assistants [2,15] for which "proving the prover" is a common expectation.

The rest of the paper is organised as follows. We first recall the main notions about coloured Petri nets. Then, we present the LLVM framework, in particular the syntax of the language and its intuitive semantics, and how it can be embedded LLVM into a Petri net as a concrete colour domain. In section 4, we present algorithms and data structures for state space exploration. We then formally define an operational semantics for LLVM, including an explicit memory model. Finally we present our correctness results. Due to the limited number of pages, many definitions and intermediary results have been omitted, as well as the detailed proofs. This material can be found in [5,4]. Notice also that our compilation approach is evaluated from a performance point of view in [6].

2 Coloured Petri Nets

A (coloured) Petri net involves objects defined by a *colour domain* that provides data values, variables, operators, a syntax for expressions, possibly typing rules, etc. Usually, elaborated colour domains are used to ease modelling; in particular, one may consider a functional programming language [9,17] or the functional fragment (expressions) of an imperative programming language [14,16]. In this paper we will consider LLVM as a concrete colour domain.

All these can be seen as implementations of a more general *abstract colour domain* providing $\mathbb{D}$ the set of *data values*, $\mathbb{V}$ the set of *variables* and $\mathbb{E}$ the set of *expressions*. Let $e \in \mathbb{E}$, we denote by $vars(e)$ the set of variables from $\mathbb{V}$ involved in e. Moreover, variables or values may be considered as (simple) expressions, *i.e.*, we assume $\mathbb{D} \cup \mathbb{V} \subseteq \mathbb{E}$.

At this abstract level, we do not make any assumption about the typing or syntactical correctness of expressions; instead, we assume that any expression can be evaluated, possibly to $\bot \notin \mathbb{D}$ (undefined value) in case of any error. More precisely, a *binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D} \cup \{\bot\}$. Then, let $e \in \mathbb{E}$ and β be a binding, we denote by $\beta(e)$ the evaluation of e under β; if the domain of β does not include $vars(e)$ then $\beta(e) \stackrel{\text{df}}{=} \bot$. The application of a binding to evaluate an expression is naturally extended to sets and multisets of expressions.

Definition 1 (Petri nets). *A* Petri net *is a tuple* (S, T, ℓ) *where* S *is the finite set of* places, T, *disjoint from* S, *is the finite set of* transitions, *and* ℓ *is a* labelling *function such that:*

- *for all* $s \in S$, $\ell(s) \subseteq \mathbb{D}$ *is the* type *of* s, i.e., *the values that* s *may contain;*
- *for all* $t \in T$, $\ell(t) \in \mathbb{E}$ *is the* guard *of* t, i.e., *a condition for its execution;*
- *for all* $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ *is a multiset over* $\mathbb{E}$ *and defines the* arc *from* x *toward* y.

A marking *of a Petri net is a map that associates to each place $s \in S$ a multiset of values from $\ell(s)$. From a marking M, a transition t can be* fired *using a binding β and yielding a new marking M', which is denoted by $M[t,\beta\rangle M'$, iff:*

- *there are enough tokens: for all $s \in S$, $M(s) \geq \beta(\ell(s,t))$;*
- *the guard is validated: $\beta(\ell(t)) = true$;*
- *place types are respected: for all $s \in S$, $\beta(\ell(t,s))$ is a multiset over $\ell(s)$;*
- *M' is M with tokens consumed and produced according to the arcs: for all $s \in S$, $M'(s) = M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$.*

Such a binding β is called a mode *of t at marking M.*

For a Petri net node $x \in S \cup T$, we define ${}^\bullet x \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(y,x) \neq \emptyset\}$ and $x^\bullet \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(x,y) \neq \emptyset\}$ where $\emptyset$ is the empty multiset. Finally, we extend the notation vars to a transition by taking the union of the variable sets in its guard and connected arcs.

In this paper, we assume that the considered Petri nets are such that, for all place $s \in S$ and all transition $t \in T$, $\ell(s,t)$ is either $\emptyset$ or contains a single variable $x \in \mathbb{V}$. We also assume that $vars(t) = \bigcup_{s \in S} vars(\ell(s,t))$, *i.e.*, all the variables involved in a transition can be bound using the input arcs. The second assumption is a classical one that allows to simplify the discovery of modes. The first assumption is made without loss of generality to simplify the presentation.

3 LLVM

The *LLVM* project (*Low Level Virtual Machine*) [11] is a modern and modular toolkit for compiler development used by a wide variety of commercial and open source projects as well as academic researches [13,12]. The *LLVM-IR* (*LLVM Intermediate Representation*) [10] is a part of the LLVM project and is a low-level, platform-independent, intermediate language. Every program written in this language can be run in a virtual machine or compiled to native code on all the platforms supported by the LLVM project. Importantly, the LLVM compiler runs a variety of optimisation passes on the LLVM-IR, which allows us to produce simple source code knowing it will be optimised by LLVM.

3.1 Syntax and Intuitive Semantics

A LLVM program is composed of a set of *blocks* (*i.e.*, sequences of instructions) identified by *labels*. Entering or leaving a block is always explicit through branching instructions (jumps), subprograms calls or return instructions.

To define the syntax, we consider the following pairwise disjoint sets:

- $\mathbb{P}$ is the set of *pointers*;
- $\mathbb{T}$ is the set of *types*, defined inductively as the smallest set containing the *primitive types* in $\mathbb{T}_0 \stackrel{\text{df}}{=} \{int, bool, \dots\}$ (integers, Boolean values and other types defined by LLVM but not needed here) and such that if $t_0, \dots, t_n \in \mathbb{T}$ then $struct(t_0, \dots, t_n) \in \mathbb{T}$, which represents a data structure with $n+1$ fields of types t_0 to t_n;

– $\mathbb{L}$ is the set of *labels*, it contains arbitrary names as well as some specific labels like $f(\!|\, a_1, \ldots, a_n \,|\!)$, where $a_i \in \mathbb{V}$ for $1 \leq i \leq n$, that correspond to subprograms entry points (including the formal parameters). We define a set $\mathbb{L}_\perp \stackrel{\text{df}}{=} \mathbb{L} \cup \{\perp\}$ where $\perp \notin \mathbb{L}$ stands for an undefined label.

A program is represented as a partial function P from $\mathbb{L}$ to the set of blocks, *i.e.*, that associates each label in its domain to a sequence of instructions.

For our purpose, we need to consider a fragment of LLVM that is formed by three main syntactic classes: sequences in *seq*, commands in *cmd* (*i.e.*, instructions) and expressions in *expr*. A sequence is a list of commands which may end with an expression, in which case it is considered as an expression itself (which is not reflected on the grammar in figure 1 to keep it simpler).

We assume that programs are syntaxically correct and well typed, so that we can simplify the syntax by forgetting all types in LLVM source code. The resulting syntax is presented in figure 1. To allow for writing one-line sequences, we introduce the sequencing operator ";" that corresponds to the line endings. We also introduce the *skip* command that denotes the empty sequence. It may be noted that *pcall* (procedure call) and *fcall* (function call) do not exist in LLVM but are different instances of the *call* instruction. This distinction can be easily made in LLVM because the instruction contains the return type of the subprogram (function or procedure). Instruction *store* (resp. *load*) is the action of storing (resp. loading) data into (resp. from) the memory through a pointer. Instruction *icmp* compares two integers. Instruction *phi* is used to access variables assigned in previously executed blocks. Instruction *gep* corresponds to pointer arithmetic, we freeze the second argument to 0, which is enough to access fields in structures by their indexes.

3.2 LLVM-Labelled Petri Nets

To compile Petri nets as defined previously into LLVM programs, we need to consider a variant where the colour domain explicitly refers to a LLVM program.

Definition 2 (LLVM labelled Petri nets). *A LLVM labelled Petri net is a tuple* $N \stackrel{\text{df}}{=} (S, T, \ell, P)$*, where* P *is a LLVM program, and such that* (S, T, ℓ) *is a coloured Petri net with the following changes:*

– *for all place* $s \in S$*,* $\ell(s)$ *is a LLVM type in* $\mathbb{T}$*, interpreted as a subset of* $\mathbb{D}$*;*
– *for all transition* $t \in T$*,* $\ell(t)$ *is a call to a Boolean function in* P *whose parameters are the elements of* $vars(t)$*;*
– *for all* $s \in t^\bullet$*,* $\ell(t, s)$ *is a singleton multiset whose unique element is a call to a* $\ell(s)$*-typed function in* P *whose parameters are the elements of* $vars(t)$*.*

We assume that all the functions involved in the annotations terminate.

With respect to the previous definition, we have concretized the place types and each expression is now implemented as a LLVM function called from the corresponding annotation. To simplify the presentation, we have also restricted the output arcs to be singleton multisets, but this can be easily generalised.

$$
\begin{array}{rll}
\mathit{seq} ::= & \mathit{cmd} & \text{(statement)} \\
| & \mathit{expr} & \text{(expression)} \\
| & \mathit{cmd};\ \mathit{seq} & \text{(sequence of instructions)} \\
\mathit{cmd} ::= & \mathit{br}\ \mathit{label} & \text{(unconditional branching)} \\
| & \mathit{br}\ \mathit{rvalue},\ \mathit{label},\ \mathit{label} & \text{(conditional branching)} \\
| & \mathit{pcall}\ \mathit{label}(\mathit{rvalue}, \ldots, \mathit{rvalue}) & \text{(procedure call)} \\
| & \mathit{ret} & \text{(return from a procedure)} \\
| & \mathit{var} = \mathit{expr} & \text{(variable assignment)} \\
| & \mathit{store}\ \mathit{rvalue},\ \mathit{rvalue} & \text{(assignment through a pointer)} \\
| & \mathit{skip} & \text{(empty sequence)} \\
\mathit{expr} ::= & \mathit{add}\ \mathit{rvalue},\ \mathit{rvalue} & \text{(addition)} \\
| & \mathit{load}\ \mathit{rvalue} & \text{(read a value through a pointer)} \\
| & \mathit{gep}\ \mathit{rvalue},\ 0,\ \mathit{nat} & \text{(get a pointer to a structure field)} \\
| & \mathit{icmp}\ \mathit{op},\ \mathit{rvalue}, \mathit{rvalue}, & \text{(integers comparison)} \\
| & \mathit{phi}\ (\mathit{rvalue}, \mathit{label}), \ldots, (\mathit{rvalue}, \mathit{label}) & \text{(get a value after branching)} \\
| & \mathit{fcall}\ \mathit{label}(\mathit{rvalue}, \ldots, \mathit{rvalue}) & \text{(function call)} \\
| & \mathit{alloc}\ \mathit{type} & \text{(memory allocation)} \\
| & \mathit{ret}\ \mathit{rvalue} & \text{(return a value from a function)} \\
| & \mathit{rvalue} & \text{(variable or value)}
\end{array}
$$

Fig. 1. Our fragment of the LLVM syntax, where $\mathit{label} \in \mathbb{L}$, $\mathit{rvalue} \in \mathbb{D} \cup \mathbb{P} \cup \mathbb{V}$, $\mathit{var} \in \mathbb{V}$, $\mathit{type} \in \mathbb{T}$, $\mathit{nat} \in \mathbb{N}$ and $\mathit{op} \in \{<, \leq, =, \neq, \geq, >\}$

Moreover, the definitions of binding and modes are extended to LLVM. A *LLVM binding* is a partial function $\beta : \mathbb{V} \to \mathbb{D} \cup \mathbb{P}$ that maps each variable from its domain to a pointer or a value, and that is widened to $\mathbb{D}$ by the identity function. $\mathbb{B}$ is the set of all LLVM bindings. A *LLVM mode* is thus a LLVM binding enabling a transition in a LLVM labelled Petri net.

4 Implementing State Space Exploration

Given an *initial marking* M_0, the state space we want to compute in this paper is the set R of *reachable marking*, *i.e.*, the smallest set such that $M_0 \in R$ and, if $M \in R$ and $M[t, \beta\rangle M'$ then $M' \in R$ also. The correctness and termination of the implementation presented in this section are addressed in section 6.

Our algorithms are implemented on the basis of data structures (multisets, places, markings, and sets) that must respect some interfaces. An interface is presented as a set of procedures or functions that manipulates a data structure through a pointer (C-like interfaces). Moreover, each such subprogram has a formal specification of its behaviour that relies on an explicit interpretation of the data structure before and after the subprogram call. Precise examples are given in section 5.3, after the definition of LLVM formal semantics.

A multiset structure to store values from a type d is assumed and we call the set of values from d having non-zero occurrences in the multiset its *domain*. The multiset interface contains in particular: two procedures $\mathit{add}_{mset}(\!| p_{mset}, \mathit{elt} |\!)$ and $\mathit{rem}_{mset}(\!| p_{mset}, \mathit{elt} |\!)$ to respectively add or remove an element elt in p_{mset}; a

function $size_{mset}(\!| p_{mset} |\!)$ to return the domain size; a function $nth_{mset}(\!| p_{mset}, n |\!)$ to return the n^{th} element from the domain (for an arbitrary fixed order).

As a container of tokens, a place can be basically implemented as a multiset of tokens. So the place interface is exactly the multiset interface but annotated by the place name, for instance add_s is like add_{mset} but for place s.

The markings interface contains for each place s a function $get_s(\!| p_{mrk} |\!)$ that returns a pointer to the corresponding place structure, as well as a function $copy_{mrk}(\!| p_{mrk} |\!)$ that returns a copy of the marking structure.

Finally, the set interface contains a function $cons_{set}(\!|\,|\!)$ that builds a new empty set and a procedure $add_{set}(\!| p_{set}, elt |\!)$ that adds an element elt to p_{set}.

Transitions Firing. Let $t \in T$ be a transition such that ${}^\bullet t = \{s_1, \ldots, s_n\}$ and $t^\bullet = \{s'_1, \ldots, s'_m\}$. Then, function $fire_t$, that computes the marking M' reachable from M by firing t given a valuation of its variables, can be expressed as shown on the left of figure 2. This function simply creates a copy M' of M, removes from it the consumed tokens and adds the produced tokens before to return M'. One could remark that it avoids a loop over the Petri net places but instead it executes a sequence of statements. This is generally more efficient (no branching penalties, no loop overhead, no lookup of functions $f_{t,s'_j}, \ldots$) and the code is simpler to generate. Let now x_{mrq} be a pointer to a marking structure implementing M. The firing algorithm can be implemented as shown on the right of figure 2.

Successors Computation. To discover all the possible modes for transition t, function $succ_t$ enumerates all the combinations of tokens from the input places. If a combination corresponds to a mode then the suitable transition firing function is called to produce a new marking. This algorithm is shown in figure 3. Note the nesting of loops that avoids an iteration over ${}^\bullet t$, which saves from querying the Petri net structure and avoids the explicit construction of a binding. Moreover, since g_t is written in the target language, we avoid an interpretation of the

$fire_t : M,\ x_1,\ \ldots,\ x_n \to M'$

$M' \leftarrow copy(M)$
$M'(s_1) \leftarrow M(s_1) - \{x_1\}$
$\ldots$
$M'(s_n) \leftarrow M(s_n) - \{x_n\}$
$M'(s'_1) \leftarrow M(s'_1) + f_{t,s'_1}(x_1, \ldots, x_n)$
$\ldots$
$M'(s'_m) \leftarrow M(s'_m) + f_{t,s'_m}(x_1, \ldots, x_n)$
return M'

$P(fire_t(\!| x_{mrq}, x_1, \ldots, x_n |\!)) \stackrel{\text{df}}{=}$

// copy the structure marking
$x'_{mrq} = fcall\ copy_{mrq}(x_{mrq})$
// consume tokens
$x_{s_i} = fcall\ get_{s_i}(x'_{mrq})$ } $1 \le i \le n$
$pcall\ rem_{s_i}(x_{s_i}, x_i)$ }
// produce tokens
$x_{s'_j} = fcall\ get_{s'_j}(x'_{mrq})$
$o_{s'_j} = fcall\ f_{t,s'_j}(x_1, \ldots, x_n)$ } $1 \le j \le m$
$pcall\ add_{s'_j}(x_{s'_j}, o_{s'_j})$
// return the new marking
$ret\ x'_{mrq}$

Fig. 2. On the left, the transition firing algorithm, where x_i is the variable in $\ell(s_i, t)$ for all $1 \le i \le n$, and f_{t,s'_j} is the function called in $\ell(t, s'_j)$ for all $1 \le j \le m$. On the right, its LLVM implementation.

corresponding expression. For the LLVM version, let x_{mrq} be a pointer to a marking structure and x_{next} be a pointer to a marking set structure. Then, the algorithm from figure 3 can be implemented as shown in figure 4. Each iteration over x_k is implemented as a set of blocks subscribed by t, k (for $n \geq k \geq 1$); blocks subscribed by $t, 0$ corresponds to the code inside the innermost loop. Note the *phi* instruction to update the value of index i_{s_i} (used to enumerate the tokens in place s_i): when the program enters block $loop_{t,i}$ for the first time, it comes from block $header_{t,i}$, so we initialise the value of i_{s_i} to the last index in the domain of s_i; later, the program comes back to block $loop_{t,i}$ from block $footer_{t,i}$, so it assigns i'_{s_i} to i_{s_i} that is the value of $i_{s_i} - 1$ (*i.e.*, the previous index).

A function *succ* is also defined to compute the set of all the successors of a marking, which is made by calling all the transition specific successor functions and accumulating the discovered markings into the same set. This algorithm and its translation in LLVM are shown in figure 5.

5 A Formal Semantics of LLVM

5.1 Memory Model

To start with, we define a memory model for LLVM, including *heaps* to store dynamically allocated pointers as well as *stacks* to store local variables and arguments for subprograms calls.

A *heap* is a partial function $H : \mathbb{P} \to \mathbb{T} \times (\mathbb{D} \cup \mathbb{P} \cup \{\bot\})^*$ with a finite domain. Each heap maps every pointer in its domain to a type and a tuple of values or pointers. The set of all heaps is $\mathbb{H}$. A heap is well formed if every pair in its image is type-consistent, for instance if $H(p) = (int, d)$ then d is indeed an integer or is $\bot$ (uninitialised).

The set of all the pointers accessible starting from a pointer p in a heap H is denoted by $p{\downarrow_H}$ and is defined for all p in $\mathbb{P}$ as:

$$
\begin{array}{ll}
p{\downarrow_H} \stackrel{\text{df}}{=} \{\} & \text{if } p \notin dom(H) \\
p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} & \text{if } H(p) = (t, v) \text{ and } t \in \mathbb{T}_0 \\
p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} \cup p_0{\downarrow_H} \cup \cdots \cup p_n{\downarrow_H} & \text{if } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n))
\end{array}
$$

In can be shown that if a heap H is well formed then $p{\downarrow_H} \subseteq dom(H)$ for every $p \in dom(H)$, and more generally that $dom(H) = \bigcup_{p \in \mathbb{P}} p{\downarrow_H}$.

Then, we need to access and update the data stored onto a heap. For each heap H, we define a data structure traversal function $\cdot[\cdot]_H : \mathbb{P} \times \mathbb{N} \to \mathbb{P} \cup \mathbb{D}$ as:

$$
p[i]_H \stackrel{\text{df}}{=} \begin{cases} p_i & \text{if } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \text{ and } 0 \leq i \leq n \\ \text{undefined} & \text{otherwise} \end{cases}
$$

The overwriting function $\oplus : \mathbb{H} \times \mathbb{H} \to \mathbb{H}$ represents the writing into memory and is defined for each $p \in \mathbb{P}$ as:

$$
(H \oplus H')(p) \stackrel{\text{df}}{=} \begin{cases} H'(p) & \text{if } p \in dom(H') \\ H(p) & \text{if } p \notin dom(H') \wedge p \in dom(H) \\ \text{undefined} & \text{otherwise} \end{cases}
$$

$succ_t : M,\ next \to \bot$

```
for x_n in M(s_n) do
  ...
  for x_1 in M(s_1) do
    if g_t(x_1, ..., x_n) then
      next ← next ∪ {fire_t(M, x_1, ..., x_n)}
    endif
  endfor
  ...
endfor
```

Fig. 3. Transition specific successors computation algorithm, where g_t is the function that evaluates the guard $\ell(t)$

$$
\begin{aligned}
P(succ_t(\!| x_{mrq}, x_{next} |\!)) &\stackrel{\text{df}}{=} \{\ br\ header_{t,n} \\
P(header_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} x_{s_k} = fcall\ get_{s_k}(x_{mrq}) \\ s_{s_k} = fcall\ size_{s_k}(x_{s_k}) \\ br\ loop_{t,k} \end{cases} \\
P(loop_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} i_{s_k} = phi\ (s_{s_k}, header_{t,k}), (i'_{s_k}, footer_{t,k}) \\ c_{s_k} = icmp\ >,\ i_{s_k},\ 0 \\ br\ c_{s_k},\ body_{t,k},\ footer_{t,k+1} \end{cases} \\
P(body_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} x_k = fcall\ nth_{s_k}(x_{s_k}, i_{s_k}) \\ br\ header_{t,k-1} \end{cases} \\
P(footer_{t,k}) &\stackrel{\text{df}}{=} \begin{cases} i'_{s_k} = add\ i_{s_k},\ -1 \\ br\ loop_{t,k} \end{cases} \\
P(header_{t,0}) &\stackrel{\text{df}}{=} \begin{cases} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{cases} \\
P(body_{t,0}) &\stackrel{\text{df}}{=} \begin{cases} x'_{mrq} = fcall\ fire_t(x_{mrq}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{next}, x'_{mrq}) \\ br\ footer_{t,1} \end{cases} \\
P(footer_{t,n+1}) &\stackrel{\text{df}}{=} \{\ ret
\end{aligned}
$$

Fig. 4. LLVM transition specific successor function, for $1 \leq k \leq n$

```
succ : M → next
next ← ∅
succ_{t_1}(M, next)
succ_{t_2}(M, next)
...
succ_{t_n}(M, next)
return next
```

$$
\begin{aligned}
&P(succ(\!| x_{mrq} |\!)) \stackrel{\text{df}}{=} \\
&\begin{cases} x_{next} = fcall\ cons_{set}() \\ pcall\ succ_{t_1}(x_{mrq}, x_{next}) \\ pcall\ succ_{t_2}(x_{mrq}, x_{next}) \\ \ldots \\ pcall\ succ_{t_n}(x_{mrq}, x_{next}) \\ ret\ x_{next} \end{cases}
\end{aligned}
$$

Fig. 5. Computation of all successors (left) and its LLVM implementation (right), where x_{mrq} is a pointer to a structure marking

In order to compare heaps, a notion of structural equivalence needs to be defined. This relation ensures that two heaps contain the same data, accessible from distinct sets of pointers but with the same layout. More precisely we consider two heaps H, H' and two pointers p, p' and write $(H,p) =_{st} (H',p')$ whenever $H(p)$ and $H'(p')$ are structurally the same values.

We also need to define an operation $new : \mathbb{H} \times \mathbb{T} \to \mathbb{H} \times \mathbb{P}$ to build new heaps, which corresponds to a pointer allocation, using a helper function $alloc : 2^{\mathbb{P}} \times \mathbb{P} \times \mathbb{T} \to \mathbb{H}$ as follows:

$$new(H,t) \stackrel{\text{df}}{=} (alloc(dom(H) \cup \{p\}, p, t), p) \qquad \text{for } p \notin dom(H) \text{ a "fresh" pointer}$$

$$alloc(d,p,t) \stackrel{\text{df}}{=} \{p \mapsto (t, \bot)\} \qquad \text{for } t \in \mathbb{T}_0$$

$$\begin{aligned} alloc(d,p,struct(t_0,\ldots,t_n)) \stackrel{\text{df}}{=}\ & \{p \mapsto (struct(t_0,\ldots,t_n),(p_0,\ldots,p_n))\} \\ & \oplus alloc(d \cup \{p_0,\ldots,p_n\}, p_0, t_0) \\ & \oplus \cdots \\ & \oplus alloc(d \cup \{p_0,\ldots,p_n\}, p_n, t_n) \\ & \qquad \text{for } p_0,\ldots,p_n \notin d \text{ "fresh" pointers} \end{aligned}$$

It can be shown that *new* always returns a well formed heap, and that calling *new* using equivalent heaps always returns equivalent heaps.

To define subprogram calls, our memory model also defines *stacks* that contain *frames* implicitly pushed onto the stack by the inference rules in the semantics. A *frame* is a tuple $F \in \mathbb{F} \stackrel{\text{df}}{=} \mathbb{L}_\bot \times \mathbb{L} \times \mathbb{B}$ whose elements are denoted by $(l_{p,F},\ l_{c,F},\ \beta_F)$, where $l_{p,F}$ is the label the block the program comes from (or undefined), $l_{c,F}$ is the label of the block currently executed, and β_F is a LLVM binding representing the current evaluation context. We widen the binding functional notation to the frames, so we denote by $F(x)$ the binding $\beta_F(x)$ of x by β_F.

Like for heaps we need operations to update frames. The same operator $\oplus$ is used because the operations are very similar, but on distinct objects. The binding overwriting operation $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ and the frame binding overwriting operation $\oplus : \mathbb{F} \times \mathbb{B} \to \mathbb{F}$ are defined as:

$$(\beta \oplus \beta')(p) \stackrel{\text{df}}{=} \begin{cases} \beta'(p) & \text{if} \quad p \in dom(\beta') \\ \beta(p) & \text{if} \quad p \notin dom(\beta') \wedge p \in dom(\beta) \\ \text{undefined otherwise} \end{cases}$$

$$(l,\ l',\ \beta) \oplus \beta' \stackrel{\text{df}}{=} (l,\ l',\ \beta \oplus \beta')$$

The structural equivalence can be widened to pairs of heaps and frames and is denoted by $(H,F) =_{st} (H',F')$ for two heaps H, H' and two frames F, F'. Intuitively, it checks that all data accessible from the frame bindings are structurally equivalent. This holds also for values stored directly in the bindings (*i.e.*, without pointers) since the heap equivalence reduces to the equality on $\mathbb{D}$.

5.2 Inference Rules

The operational semantics is defined for a fixed and immutable program P, which means that no function nor block can be created nor modified during the execution. We denote the result of a computation by $\overline{\cdot}$, for example $\overline{2+3}$ is 5. The main objects handled by our inference rules are *configurations* that represent a state of the program during its execution. A configuration is a tuple (seq, H, F), denoted by $\left(seq\right)_{H,F}$, where seq is a sequence of instructions, H is a heap and F is a frame.

The inference rules for expressions are shown in figure 6; expressions evaluate to values in the context of a frame. The inference rules for sequence and commands are shown in figure 7; sequences and commands evaluate to other sequences or commands in the context of a heap and a frame. One can remark how a frame is pushed onto the stack in *pcall* and *fcall* rules, a new frame F_0 is actually replacing the current frame F in the subsumption of these rules and used to execute the body of the called subprogram. This semantics mixes up small-step and big-step reductions. Indeed, most of the rules are small-step except for *pcall* and *fcall* rules in which we link the computation to its result by making a sequence of reductions in the rule subsumption.

5.3 Data Structures Interpretation

The link between Petri nets and their LLVM implementation is formalised with a family of interpretation functions for all data structures. This allows to formalise the behavioural requirements on the interfaces presented in section 4.

An interpretation is a partial function which maps a pair formed by a heap and a pointer to a Petri net object: a marking, a set of markings, a multiset of tokens or a single token, depending on the interpreted object. Interpretations are denoted by $[\![H, p]\!]^\star$, where $H \in \mathbb{H}$, $p \in \mathbb{P} \cup \mathbb{D}$ and $\star$ is an annotation describing the interpreted object (for instance we use $mset(t)$ instead of $\star$ to interpret a multiset over a type t). Whenever p is a pointer, we assume that the interpretation depends only on data that is accessible from p, *i.e.*, $p{\downarrow_H}$. Moreover every interpretation function has to respect the following consistency requirement.

Requirement 1 (Consistency) . *Let H, H' be two heaps, $[\![\cdot,\cdot]\!]^\star$ an interpretation function, and p, p' two pointers or values. If $(H, p) =_{st} (H', p')$ then $[\![H, p]\!]^\star = [\![H', p']\!]^\star$.*

As presented in section 4, we use data structures and functions as basic blocks for constructing our algorithms, they are either predefined or produced by the compilation process. Each of these functions and data structures is specified (actually, axiomatized) by a formal interface. In particular, this helps to ensure independence and modularity between components both in a programmatic and formal way. Specifying an interface leads to define a set of primitives that respect given derivations and interpretations. For example, let H be a heap and F a frame such that, $F(x_{mset}) = p_{mset}$ is a pointer on a multiset structure storing

$$\frac{F(x) = p \quad H(p) = (t, v)}{\left(load\ x \right)_{H,F} \rightsquigarrow \left(v \right)_H}\ load \qquad \frac{F(x) = p \quad p[i]_H \text{ is defined}}{\left(gep\ x,\ 0,\ i \right)_{H,F} \rightsquigarrow \left(p[i]_H \right)_H}\ gep_0$$

$$\frac{\overline{F(x_1) + F(x_2)} = v}{\left(add\ x_1,\ x_2 \right)_{H,F} \rightsquigarrow \left(v \right)_H}\ add \qquad \frac{(H', p) = new(H, t)}{\left(alloc\ t \right)_{H,F} \rightsquigarrow \left(p \right)_{H \oplus H'}}\ alloc$$

$$\frac{\overline{F(x_1)\ op\ F(x_2)} = v \quad op \in \{<, \leq, =, \neq, \geq, >\}}{\left(icmp\ op,\ x_1,\ x_2 \right)_{H,F} \rightsquigarrow \left(v \right)_H}\ icmp$$

$$\frac{1 \leq i \leq n \quad l_i \neq \bot}{\left(phi\ (x_1, l_1), \ldots, (x_n, l_n) \right)_{H,(l_i,\ l_c,\ \beta)} \rightsquigarrow \left(\beta(x_i) \right)_H}\ phi$$

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot,\ f(\!|\, a_1, \ldots, a_n \,|\!),\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(P(f(\!|\, a_1, \ldots, a_n \,|\!)) \right)_{H,F_0} \rightsquigarrow^* \left(v \right)_{H'} \end{array}}{\left(fcall\ f(r_1, \ldots, r_n) \right)_{H,F} \rightsquigarrow \left(v \right)_{H'}}\ fcall$$

$$\frac{}{\left(ret\ r \right)_{H,F} \rightsquigarrow \left(F(r) \right)_H}\ ret$$

Fig. 6. Rules for expressions

$$\frac{\left(cmd \right)_{H,F} \rightsquigarrow \left(seq' \right)_{H',F'}}{\left(cmd;\ seq \right)_{H,F} \rightsquigarrow \left(seq';\ seq \right)_{H',F'}}\ seq$$

$$\frac{}{\left(skip;\ seq \right)_{H,F} \rightsquigarrow \left(seq \right)_{H,F}}\ skip$$

$$\frac{}{\left(br\ l \right)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \left(P(l) \right)_{H,(l_c,\ l,\ \beta)}}\ branch_1$$

$$\frac{(\beta(r) = true \wedge l = l_1) \vee (\beta(r) = false \wedge l = l_2)}{\left(br\ r,\ l_1,\ l_2 \right)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \left(P(l) \right)_{H,(l_c,\ l,\ \beta)}}\ branch_2$$

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot,\ f(\!|\, a_1, \ldots, a_n \,|\!),\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(P(f(\!|\, a_1, \ldots, a_n \,|\!)) \right)_{H,F_0} \rightsquigarrow^* \left(ret \right)_{H',F'} \end{array}}{\left(pcall\ f(r_1, \ldots, r_n) \right)_{H,F} \rightsquigarrow \left(skip \right)_{H',F}}\ pcall$$

$$\frac{\left(expr \right)_{H,F} \rightsquigarrow \left(v \right)_{H'}}{\left(x = expr \right)_{H,F} \rightsquigarrow \left(skip \right)_{H',F \oplus \{x \mapsto v\}}}\ assign$$

$$\frac{F(r_p) = p \quad H(p) = (t, d) \quad H' = \{p \mapsto (t, F(r_{new}))\}}{\left(store\ r_{new},\ r_p \right)_{H,F} \rightsquigarrow \left(skip \right)_{H \oplus H',F}}\ store$$

Fig. 7. Rules for sequences and commands

elements of type t. Under these conditions, procedure add_{mset} is specified by:

$$\left(\, pcall\ add_{mset}(x_{mset}, x)\,\right)_{H,F} \rightsquigarrow \left(\, skip\,\right)_{H \oplus H', F} \tag{1}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H} \tag{2}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H, p_{mset}]\!]^{mset(t)} + \{[\![H, F(x)]\!]^t\} \tag{3}$$

Condition (1) describes by a reduction the result of the call, condition (2) restrict the updates to be localised in the heap and condition (3) interprets the computation in terms of Petri nets objects. Similarly, any implementation of the marking structures has to respect the two following requirements.

Requirement 2 (Soundness). *Let H, H' be two heaps, F, F' two frames, $p_{mrq} \in dom(H)$ a pointer to a marking structure, and p_s a pointer to a place nested in p_{mrq}* (i.e., *$p_s \in p_{mrq}{\downarrow_H}$*). *If $[\![H, p_{mrq}]\!]^{mrq}(s) = [\![H, p_s]\!]^s$, $\left(\, seq\,\right)_{H,F} \rightsquigarrow \left(\, seq'\,\right)_{H \oplus H', F'}$ and $p_{mrq} \notin dom(H')$ then*

$$[\![H \oplus H', p_{mrq}]\!]^{mrq}(s) = [\![H \oplus H', p_s]\!]^s$$

Requirement 3 (Separation). *Let p_{mrq} be a pointer on a structure marking in a heap H. If p_s and $p_{s'}$ are pointers to distinct places in this structure then we have $p_s{\downarrow_H} \cap p_{s'}{\downarrow_H} = \emptyset$.*

The *soundness* property ensures that any update of a place through a pointer returned by get_s is actually made on the marking (not on a copy). The *separation* property ensures that places do not share memory so that updating a place does not have side effects on other places.

6 Correctness and Termination Results

We present now the two main results proving the correctness of functions $fire_t$ (theorem 1) and $succ_t$ (theorem 2). Both these results are shown in a minimal context, *i.e.*, a heap that just contains the required pointers. An auxiliary theorem (not presented here) allows to generalise both results to any context that includes the minimal one.

Theorem 1. *Let M be a marking, H a heap and p_{mrq} a pointer on a marking structure such that $dom(H) = p_{mrq}{\downarrow_H}$ and $[\![H, p_{mrq}]\!]^{mrq} = M$. Let $\beta \stackrel{\text{df}}{=} \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ be a LLVM mode for transition t, which implies that each v_i is a value or a pointer encoding a token in place s_i: $[\![H, v_i]\!]^{\ell(s_i)} \in M(s_i)$. Let F be a frame such that $\beta_F \stackrel{\text{df}}{=} \beta \oplus \{x_{mrq} \mapsto p_{mrq}\}$. If*

$$M[t, \beta\rangle M' \quad \text{and} \quad \left(\, fcall\ fire_t(x_{mrq}, x_1, \ldots, x_n)\,\right)_{H,F} \rightsquigarrow \left(\, p'_{mrq}\,\right)_{H \oplus H'}$$

then

$$[\![H \oplus H', p'_{mrq}]\!]^{mrq} = M' \quad \text{and} \quad dom(H) \cap dom(H') = \emptyset$$

Proof (sketch). This theorem is the direct application of two lemmas showing the correctness of tokens consumption and production respectively. Both are proved by induction on the number of places in the marking structure. □

Corollary 1. *Under the same hypothesis, the call to* $fire_t$ *terminates.* □

Theorem 2. *Let* F *be a frame and* p_{mrq} *a pointer in a heap* H *such that* $p_{mrq}{\downarrow_H} = dom(H)$, $[\![H, p_{mrq}]\!]^{mrq} = M$, $\beta_F(x_{mrq}) = p_{mrq}$ *and* $\beta_F(x_{next}) = p_{next}$. *If*

$$\left(fcall\ succ_t(x_{mrq}, x_{next})\right)_{H,F} \rightsquigarrow^* \left(p_{next}\right)_{H \oplus H'} \quad and \quad [\![H, p_{next}]\!]^{set} = E$$

where E *si a set of markings, then*

$$dom(H) \cap dom(H') = \emptyset$$

$$[\![H \oplus H', p_{next}]\!]^{set} = E \cup \{M' \mid \exists \beta,\ M[t, \beta\rangle M'\}$$

Proof (sketch). The first result is a consequence of the formal interfaces of the called functions. The second result is proved as two inclusions:

$\supseteq$. This is the consequence of two lemmas:
- by applying reduction rules, we show that if execution goes through a block $header_{t,k}$ then it will necessarily reach a block $footer_{t,k+1}$, and every block annotated by an index greater than k must be executed also;
- consequently, all combinations of tokens for the input places are actually enumerated, which implies that all potential modes of t are considered.

$\subseteq$. To prove that only actual successor markings are added, we first remark that if a marking is added into the set, then this happens in block $body_{t,0}$. So it is enough to prove that this block is executed only if the binding is actually a mode for t, which can be proved using the reduction rules backward to show that the guard necessarily evaluated to *true*. □

Corollary 2. *Under the same hypothesis, the call to* $succ_t$ *terminates.* □

7 Conclusion

We have shown how a Petri net can be compiled targeting a fragment of the LLVM language. This compilation produces code that provides the primitives to compute the state space of the compiled Petri net model. Then we have defined a formal semantics for the fragment of the LLVM language we use. To produce a readable and usable system of inference rules, we have defined a memory model based on explicit heaps and stacks. Finally we have proved the correctness of the code generated by our compiler. The full proofs provided in [5,4] are quite long because they are very much detailed to improve our confidence into their correctness and to ease there later validation using a proof assistant. But notice

also that they are at the same time quite easy to follow. It is worth noting also that our proofs are modular thanks to clearly defined interfaces with appropriate axiomatisation. As a consequence, we should avoid issues when parts of the generated code are replaced, for instance, to use a more efficient data structure, or alternative state space exploration approaches (like in [8]).

The fragment of LLVM we have considered is rather limited with respect to the number of instructions. However, it is at the same time quite representative of the full language. Indeed, it includes the necessary to handle the stack and the heap which are conceptually the most complicated parts of the language. Extending our fragment to include all the LLVM computational instructions (like arithmetic) would be an easy but tedious work. Adding the instructions to manipulate the stack (like *unwind* for exception handling) looks quite straightforward. The most complicated is probably adding full support for pointers, which would required to refine our heap model (in particular, pointer arithmetic would have to be defined).

In this paper, we have considered an "optimistic" approach in that we assume that the LLVM code provided in the model annotations is correct and terminates. Moreover, we did not make any assumption about the finiteness of the state space or the boundedness of integer values that are assumed not to overflow. In practice, this are however important issues. Approaches based on abstract interpretation of assembly code like [18] may be helpful to prove such properties on the compiled model before to start the state space exploration, ensuring that it will run safely (and allowing to avoid implementing checks in the generated code).

Future works will address a generalisation of the presented approach to compile a wider variety of coloured Petri nets, in particular nets embedding annotation languages easier to use for the modeller than LLVM. Moreover, we would like to refine requirement 3 to allow for a logical separation instead of a physical separation as it is currently defined. This would enable us for implementing memory sharing and thus saving a lot of memory during a state space exploration. We are also interested in particular in exploiting remarkable structures of Petri net models that allow to optimise the code generated by the compiler. Such optimisations also need to be formally proved and preliminary results about this can be found in [4]. A longer term goal is to prove the whole compilation chain to obtain the core (*i.e.*, state space exploration) of a certified explicit model-checker for coloured Petri nets. A complementary aspect is to evaluate the performance of the state space generation, which is of course another important motivation when working on a model-checker. As shown in [6], the current implementation is efficient and can outperform state-of-the-art tools. So, certification is not an objective that contradicts efficiency.

References

1. Clarke, E., Emerson, A., Sifakis, J.: Model checking: Algorithmic verification and debugging. ACM Turing Award (2007)
2. ADT Coq/INRIA. The Coq proof assistant, http://coq.inria.fr

3. Evangelista, S.: Méthodes et outils de vérification pour les réseaux de Petri de haut niveau. PhD thesis, CNAM, Paris, France (2006)
4. Fronc, L.: Analyse efficace des réseaux de Petri par des techniques de compilation. Master's thesis, MPRI, university of Paris 7 (2010), http://www.ibisc.fr/~lfronc/pub/LF_2010.pdf
5. Fronc, L., Pommereau, F.: Proving a Petri net model-checker implementation. Technical report, IBISC, University of Évry (2010), http://goo.gl/WMzhp
6. Fronc, L., Pommereau, F.: Optimizing the compilation of Petri nets models. In: Proc. of SUMo 2011. CEUR, vol. 726 (2011), http://ceur-ws.org/Vol-726
7. Holzmann, G.J., et al.: Spin, formal verification, http://spinroot.com
8. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth-first search. In: Proc. of the 2nd Spin Workshop. AMS (1996)
9. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009); ISBN: 978-3-642-00283-0
10. Lattner, C.: LLVM language reference manual, http://llvm.org/docs/LangRef.html
11. Lattner, C., et al.: The LLVM compiler infrastructure., http://llvm.org
12. Lattner, C., et al.: LLVM related publications., http://llvm.org/pubs
13. Lattner, C., et al.: LLVM users, http://llvm.org/Users.html
14. Pajault, C., Evangelista, S.: Helena: a high level net analyzer, http://helena.cnam.fr
15. Paulson, L., Nipkow, T., Wenzel, M.: The Isabelle proof assistant, http://www.cl.cam.ac.uk/research/hvg/Isabelle
16. Pommereau, F.: Quickly prototyping Petri nets tools with SNAKES. Petri net newsletter (2008)
17. Reinke, C.: Haskell-coloured Petri nets. In: Koopman, P., Clack, C. (eds.) IFL 1999. LNCS, vol. 1868, pp. 181–198. Springer, Heidelberg (2000)
18. Xavier Rival. Traces Abstraction in Static Analysis and Program Transformation Abstraction de Traces en Analyse Statique et Transformations de Programmes. PhD thesis, Computer Science Department, École Normale Supérieure (2005)
19. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting bDDs in coq. In: Kleinberg, R.D., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 162–181. Springer, Heidelberg (2000)

Elementary Linear Logic Revisited for Polynomial Time and an Exponential Time Hierarchy

Patrick Baillot

ENS Lyon, Université de Lyon, LIP (UMR 5668 CNRS-ENSL-INRIA-UCBL)
patrick.baillot@ens-lyon.fr

Abstract. Elementary linear logic is a simple variant of linear logic, introduced by Girard and which characterizes in the proofs-as-programs approach the class of elementary functions, that is to say computable in time bounded by a tower of exponentials of fixed height.

Our goal here is to show that despite its simplicity, elementary linear logic can nevertheless be used as a common framework to characterize the different levels of a hierarchy of deterministic time complexity classes, within elementary time. We consider a variant of this logic with type fixpoints and weakening. By selecting specific types we then characterize the class P of polynomial time predicates and more generally the hierarchy of classes $\mathsf{k\text{-}EXP}$, for $k \geq 0$, where $\mathsf{k\text{-}EXP}$ is the union of $\mathsf{DTIME}(2_k^{n^i})$, for $i \geq 1$.

1 Introduction

Implicit Computational Complexity. This line of research promotes investigations to delineate classical complexity classes by programming languages or logics, without refering to explicit bounds on resources (time, space ...) but instead by restricting the primitives or the features of the languages. Various approaches have been used for that, primarily in logic and in functional languages: restrictions of the comprehension scheme in second-order logic [Lei91, Lei02]; ramification in logic or in recursion [Lei94, BC92]; *read-only* functional programs [Jon01]; variants of linear logic [Gir98, Laf04]... to name only a few.

Note that the programming disciplines induced by these systems are quite restrictive, but some of these characterizations have in a second step led to more flexible criteria for statically checking complexity bounds on programs: ramification and safe recursion have inspired the work on interpretation methods for complexity [BMM11] on the one-hand, and linear type systems for non-size-increasing computation [Hof03] on the other, which itself has led to typing methods for amortized complexity analysis [HJ06, HAH11].

Linear Logic. The linear logic approach to implicit complexity fits in the proofs-as-programs paradigm. It stems from the observation that as duplication is controlled in linear logic by the modality !, weaker versions of this modality can define systems with a complexity-bounded normalization procedure: *elementary*

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 337–352, 2011.

linear logic (ELL) [Gir98, DJ03] characterizes in this way the class of Kalmar elementary functions (computable in time bounded by a tower of exponentials of fixed height), while light linear logic (LLL) [Gir98] and soft linear logic (SLL) [Laf04] characterize functions computable in polynomial time. These logical systems have then enabled the design of type systems for λ-calculus or functional languages ensuring that a well-typed program has a polynomial time complexity bound [BT09, GR07, BGM10].

Note that initially ELL does not sound as interesting as LLL and SLL since it corresponds to elementary complexity, which is not very relevant from a programming point of view. However it has nice logical properties, a simpler language of formulas than LLL (no § modality) and allows for a more natural programming style than SLL. It has also been studied for its remarkable properties concerning λ-calculus optimal reduction [ACM04].

Calibrating Complexity. One might however deplore a lack of homogeneity in these characterizations of complexity classes by variants of linear logic: indeed some common deterministic complexity classes like EXP have not yet been characterized, a different system is needed for each complexity class, and these various systems are not easy to compare.

By contrast, other methods in implicit complexity have provided frameworks that can be calibrated to delineate different complexity classes inside the (large) Kalmar elementary class:

- Jones considers in [Jon01] a *read-only* functional programming language and characterizes in it the classes $\mathsf{k\text{-}EXP}$, for $k \geq 0$, by considering, for each k, programs using only arguments of type-order at most k;
- Leivant investigates in [Lei02] second-order logic with comprehension (quantifier elimination) restricted to various families of first-order formulas: the functions provably total in this logic with comprehension restricted to formulas of type-order at most k are precisely the functions of $\mathsf{k\text{-}EXP}$.

Note that even if these two frameworks fit in different computational approaches, they both use as parameter the type-order (or implicational rank) of formulas.

Objective and Contribution. A goal of the present work is to provide an analogous framework in linear logic, allowing to characterize in a single logic a hierarchy of complexity classes by calibrating a certain parameter. We will use elementary linear logic, which offers the advantage of simplicity. A key parameter in this system, as in LLL, is the number of nested modalities (!) and this will be the value calibrating our complexity bounds.

For technical reasons we will actually consider an extension of ELL obtained by adding to it type fixpoints. It had been observed from the beginning [Gir98] that this feature does not modify the dynamics of ELL and its complexity bounds. We will also allow unrestricted weakening, which is innocuous and common practise since [AR02]. Given some types $\mathbf{W}$ for binary words and $\mathbf{B}$ for booleans, we will then show that for $k \geq 0$, the proofs of conclusion $!\mathbf{W} \multimap !^{k+2}\mathbf{B}$, where $!^i$ stands for a sequence of i !s, correspond to the predicates of $\mathsf{k\text{-}EXP}$. In particular this gives, in the case where $k = 0$, a characterization of the class P in an elementary logic, with the type $!\mathbf{W} \multimap !^2\mathbf{B}$.

Note that a distinctive point of our characterization is that it does not rely on a restriction of a particular operation *inside* the proof-program, like the application of a function to an argument in [Jon01] or the comprehension rule in [Lei02]. Instead it only imposes a condition on the *conclusion* of the proof (or type of the program), that its to say on its interface. In this sense it is more modular than these previous characterizations. Observe also that our system is a second-order logic, as the one of [Lei02], but here comprehension is not restricted.

Because of space constraints we had to omit some proofs; they can be found in [Bai11].

2 Characterization of the Classes P and k-EXP

We consider intuitionistic affine elementary logic with type fixpoints that we denote by EAL_μ. Actually for our purpose it is sufficient to consider its multiplicative fragment. The grammar of types is:

$$A ::= \alpha \mid A \multimap A \mid A \otimes A \mid !A \mid \forall \alpha.A \mid \mu\alpha.A$$

We will represent functions by proofs, but as often it will be convenient to use λ-calculus to denote the algorithmic content of proofs. For that we will consider an extension of λ-calculus with a $\otimes$ construction:

$$t, u ::= x \mid \lambda x.t \mid (t\ u) \mid t \otimes u \mid \mathrm{let}\, t\, \mathrm{be}\, x \otimes y\, \mathrm{in}\, u$$

Its reduction rule is obtained by the context-closure of the usual β-reduction rule and of the following one:

$$\mathrm{let}\, t_1 \otimes t_2\, \mathrm{be}\, x \otimes y\, \mathrm{in}\, u \to u[t_1/x, t_2/y].$$

The rules of EAL_μ are now given on Fig. 1, as a sequent calculus decorated with λ-terms. This system only differs from the intuitionistic version of elementary linear logic without additive connectives [Gir98, DJ03] by the fact that we have added the fixpoint construction (rules L_μ and R_μ) and allowed for general weakening (rule (Weak)). Observe that some rules do not have any effect on the term (!, L_μ, R_μ, $L_\forall$, $R_\forall$): this is because we want to keep the term calculus as simple as possible, and the current calculus is anyway sufficient to represent the functions denoted by the proofs.

Observe that the formulas $!A \multimap A$ and $!A \multimap !!A$ are *not* provable, which is the distinctive feature of elementary linear logic with respect to ordinary linear logic.

Finally, note that if we added the fixpoint rules to intuitionistic logic or linear logic cut elimination would not be normalizing anymore, but strong normalization does hold for EAL_μ (see [Gir98]).

Let us denote the following types respectively for booleans, n-ary finite types, tally integers and binary words, which are adapted from system F:

$$\mathbf{B} = \forall\alpha.\alpha \multimap \alpha \multimap \alpha, \quad \mathbf{B^n} = \forall\alpha.\alpha \multimap \ldots \multimap \alpha, \text{ with } n+1 \text{ occurrences of } \alpha$$
$$\mathbf{N} = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha), \quad \mathbf{W} = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$$

Axiom and Cut.

$$\frac{}{x : A \vdash x : A}\ Ax \qquad \frac{\Gamma \vdash t : A \quad \Delta, x : A \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}\ Cut$$

Structural Rules.

$$\frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t : A}\ Weak \qquad \frac{\Gamma, x_1 :!A, x_2 :!A \vdash t : B}{\Gamma, x :!A \vdash t[x/x_1, x/x_2] : B}\ Contr$$

Multiplicative Rules.

$$\frac{\Gamma \vdash t : A \quad \Delta, x : B \vdash u : C}{\Gamma, \Delta, y : A \multimap B \vdash u[(y\ t)/x] : C}\ L_{\multimap} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ R_{\multimap}$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash t : C}{\Gamma, x : A \otimes B \vdash \mathrm{let}\, x\, \mathrm{be}\, x_1 \otimes x_2\, \mathrm{in}\, t : C}\ L_{\otimes} \qquad \frac{\Gamma \vdash t_1 : A \quad \Delta \vdash t_2 : B}{\Gamma, \Delta \vdash t_1 \otimes t_2 : A \otimes B}\ R_{\otimes}$$

Exponential Logical Rule.

$$\frac{\Gamma \vdash t : A}{!\Gamma \vdash t :!A}\ !$$

Second Order Rules

$$\frac{\Gamma, x : C[A/\alpha] \vdash t : B}{\Gamma, x : \forall\alpha.C \vdash t : B}\ L_{\forall} \qquad \frac{\Gamma \vdash t : C \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash t : \forall\alpha.C}\ R_{\forall}$$

Fixpoint Rules

$$\frac{\Gamma, x : A[\mu\alpha.A/\alpha] \vdash t : B}{\Gamma, x : \mu\alpha.A \vdash t : B}\ L_{\mu} \qquad \frac{\Gamma \vdash t : A[\mu\alpha.A/\alpha]}{\Gamma \vdash t : \mu\alpha.A}\ R_{\mu}$$

Fig. 1. The system EAL_{μ}

Recall that these data-types admit some coercions [Gir98]. For $\mathbf{W}$ for instance, one can give a proof of type $\mathbf{W} \multimap !\mathbf{W}$ which, as a λ-term, acts as an identity on the terms encoding binary words.

We denote: $2_0^n = n$, $2_{k+1}^n = 2^{2_k^n}$. Recall that elementary functions are the functions computable on a Turing machine in time $O(2_k^n)$, for some k.

The standard results of elementary linear logic [DJ03] still hold in the setting of EAL_{μ}:

- for any integer d, any proof of $\mathbf{N} \multimap !^d\mathbf{N}$ represents an elementary function (*complexity soundness*);
- for any elementary function $f : \mathbb{N} \to \mathbb{N}$ there exists an integer d and a proof of $\mathbf{N} \multimap !^d\mathbf{N}$ representing it (*extensional completeness*).

Now, by considering alternative types we will be able to delineate more complexity classes:

Theorem 1. *We consider the system EAL_{μ}.*

- *The functions representable by proofs of* $!\mathbf{W} \multimap !^2\mathbf{B}$ *(resp.* $!\mathbf{W} \multimap !^3\mathbf{B}$*) are exactly the class* P *(resp.* EXP*);*
- *More generally, for any* $k \geq 0$*, the functions representable by proofs of conclusion* $!\mathbf{W} \multimap !^{k+2}\mathbf{B}$ *are exactly the class* k-EXP*, where:* $\mathsf{k\text{-}EXP} = \cup_{i\in\mathbb{N}}\mathsf{DTIME}(2_k^{n^i})$, $\mathsf{EXP} = \mathsf{1\text{-}EXP}$.

Remark 1. Note that we do not use fixpoints in the final types involved. However, technically speaking the fixpoints are used in the proofs of completeness, in order to simulate polynomial time (resp. k-exponential time) Turing machines, as we will see in Sect. 4.

Observe that as in [Jon01] we are characterizing here predicates and not general functions. We will come back to this point later, in Remark 5.

In the rest of the paper we will prove Theorem 1: Sect. 3 will establish the complexity soundness (proofs with these types represent predicates of the given complexity classes) and Sect. 4 will prove the extensional completeness (all predicates of these complexity classes can be computed by suitable proofs).

3 Proof-Nets and Complexity Soundness

In order to prove the complexity bound we study the cut elimination process and take advantage of the assumption that the conclusion of the proof is of type $!\mathbf{W} \multimap !^{k+2}\mathbf{B}$ in order to derive a sharper bound. For that, as usual in linear logic it is convenient to use *proof-nets* to analyse cut elimination as proof-net reduction. The proof-nets will use formulas of classical elementary linear logic with fixpoints:

$$A ::= \alpha \mid \alpha^\perp \mid A \otimes A \mid A \wp A \mid !A \mid ?A \mid \mu\alpha.A \mid \overline{\mu}\alpha.A \mid \forall\alpha.A \mid \exists\alpha.A$$

The connectives (modalities) ! and ? are called *exponentials* and $\otimes/\wp$ are *multiplicatives*.

Formulas of EAL_μ are translated in this grammar by using $A \multimap B \equiv A^\perp \wp B$ and the usual linear logic De Morgan laws for linear negation:

$$(A \otimes B)^\perp \equiv A^\perp \wp B^\perp, \qquad (!A)^\perp \equiv ?A^\perp,$$
$$(\mu\alpha.A)^\perp \equiv \overline{\mu}\alpha.A^\perp, \quad (\forall\alpha.A)^\perp \equiv \exists\alpha.A^\perp, \qquad A^{\perp\perp} \equiv A.$$

In order to handle weakening we will use proof-nets with polarities, following [AR02]. Note that proof-nets with polarities had been considered before, e.g. in [Lam96], but here we will follow the conventions and notations of [AR02]. The nodes are described on Fig. 2:

- nodes have ports which are positive (dark bullet) or negative (white bullet); an edge can link together either two positive or two negative ports; we say that an edge is positive (resp. negative) if it is connected to a positive (resp. negative) port;

– as the system is affine, the proof-nets use, beside the weakening w-node, also an h-node;
– two nodes μ and $\overline{\mu}$ are added, corresponding resp. to the fixpoint rules R_μ and L_μ;
– each node ! or ? comes with a !-box, as shown on Fig. 3; two boxes are either disjoint or one is included in the other; the !-node is called the *principal door* of the box and the ?-nodes are its *auxilliary doors*.

One will translate an EAL_μ proof π of conclusion $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ by a graph π^* with conclusions ${A_1}^\perp, \ldots, {A_n}^\perp, B$ where the edges of the ${A_i}^\perp$ $(1 \leq i \leq n)$ are negative and the edge of B is positive. This translation is standard [Gir87] and we only describe a couple of illustrative cases:

– if π is obtained by an Ax rule, then π^* is an ax-node;
– if π is obtained by a Cut rule between π_1 and π_2, then π^* is obtained by linking ${\pi_1}^*$ and ${\pi_2}^*$ by a *cut*-node;
– if π is obtained by a !-rule on π_1, then π^* is obtained by applying a !-box on ${\pi_1}^*$ (see Fig. 3),

and so on. On Fig. 2 below each node we have indicated the sequent calculus rule it corresponds to. Note that the h-node is not used for this translation; it will only appear during normalisation. Now, a graph R is called a *proof-net* if there exists a proof π such that $R = \pi^*$.

A *cut*-node between an ax-node (resp. w-node) and another node (resp. another node which is not an ax-node) is called an *axiom cut* (resp. a *weakening cut*). A cut between a $\otimes$-node and a $\wp$-node is called a *multiplicative cut*. A cut between a !-node and a ?-node or ?c-node (resp. a ?c-node) is called an *exponential cut* (resp. a contraction cut). *Quantifier cuts* and *μ-cuts* are defined in an analogous way.

In a proof-net, a maximal tree with ?-nodes, ax-nodes and w-nodes (of type $?A^\perp$) as leaves, and ?c-nodes as internal nodes, is called an *exponential tree*.

Now, given R, the *depth* of a node is the number of exponential boxes containing it, and the depth $d(R)$ of R is the maximal depth of its nodes. Let $|R|_i$ denote the number of nodes at depth i which are not cut nodes, w-nodes or h-nodes. We denote $|R| = \Sigma_{i=0}^{d(R)} |R|_i$.

Reduction. The reduction procedure on proof-nets consists in eliminating cuts. Because of space constraints we do not give here all reduction rules, but only the most important ones, described on Fig. 4, 5, 6:

– Fig. 5 shows the reduction of exponential cuts (contraction step and box-box step);
– Fig. 6 gives a sample of reduction rules for weakening cuts (notice that one of these steps introduces h-nodes) and cuts on h-nodes; the remaining rules are similar and can be found in [Bai11].

Observe that during a reduction step the depth of an edge does not change, hence the depth of the proof-net does not increase. This is called the *stratification*

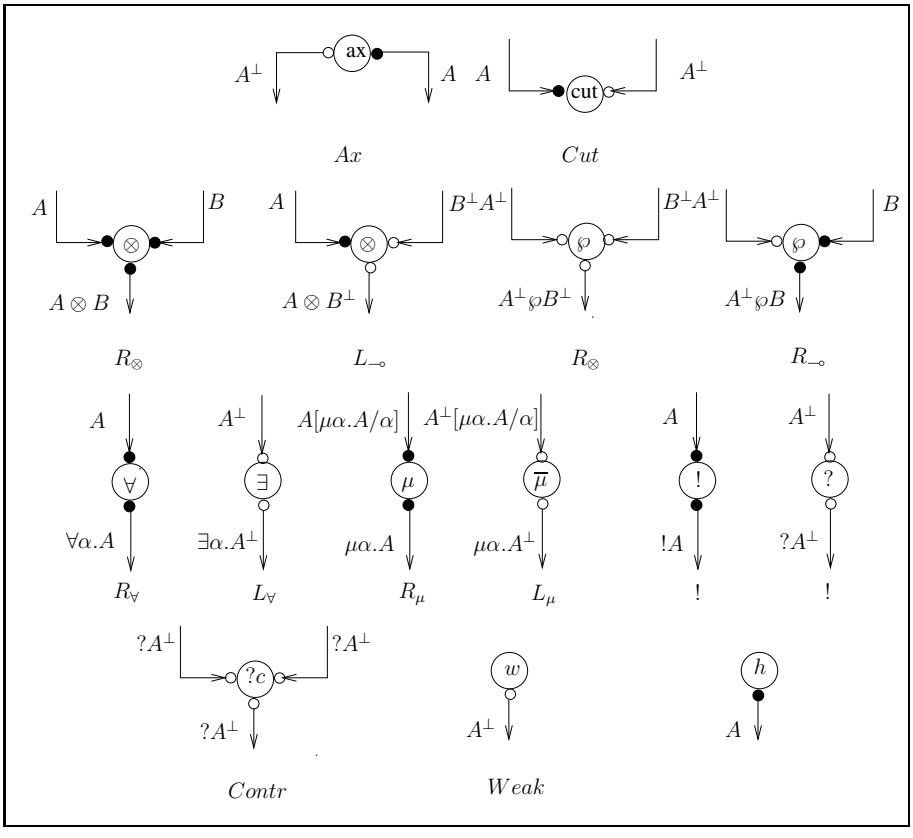


Fig. 2. Nodes of the proof-nets

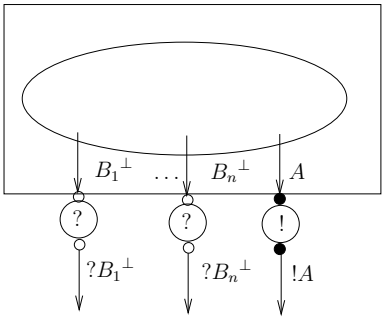


Fig. 3. !-box

property [DJ03] and it is a key ingredient for the complexity properties. It is not valid in ordinary linear logic, and it comes from the fact that during reduction a !-box is not opened and does not enter another box (see Fig. 5).

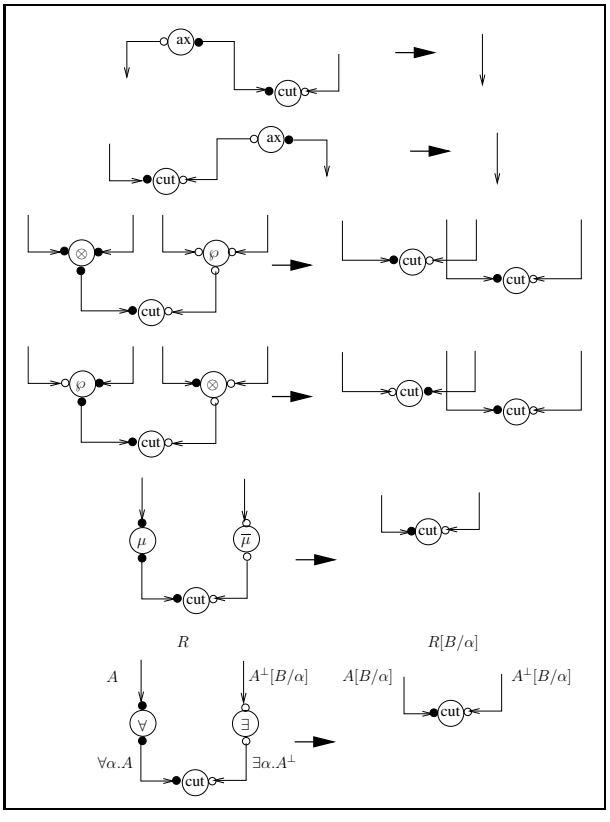


Fig. 4. Reduction steps (1/3)

We say that an exponential cut c is a *special cut* if the box $\mathcal{B}$ corresponding to the ! node does not have any cut below its auxilliary doors. The following fact can be easily verified by examining each reduction step other than the contraction reduction step:

Lemma 1. *Let R be a proof-net and R' be obtained from R by reducing a cut at depth i which is not a contraction cut. Then we have $|R'|_i < |R|_i$ and $|R'| < |R|$.*

Now let us briefly recall the method used in [Gir98] to establish complexity bounds on proof-net reduction in light linear logic and elementary linear logic. It uses a specific reduction strategy, obtained by the two following ideas:

- reduce the cuts level-by-level, that is to say first at depth i (*round i*) for i successively equal to $0, 1, \ldots, d(R)$;
- at a given depth i, proceed in two phases:
  - first reduce cuts that make the size decrease, so in the case of ELL all cuts but the exponential cuts,
  - then reduce the exponential cuts, by repeatedly reducing a *special cut*.

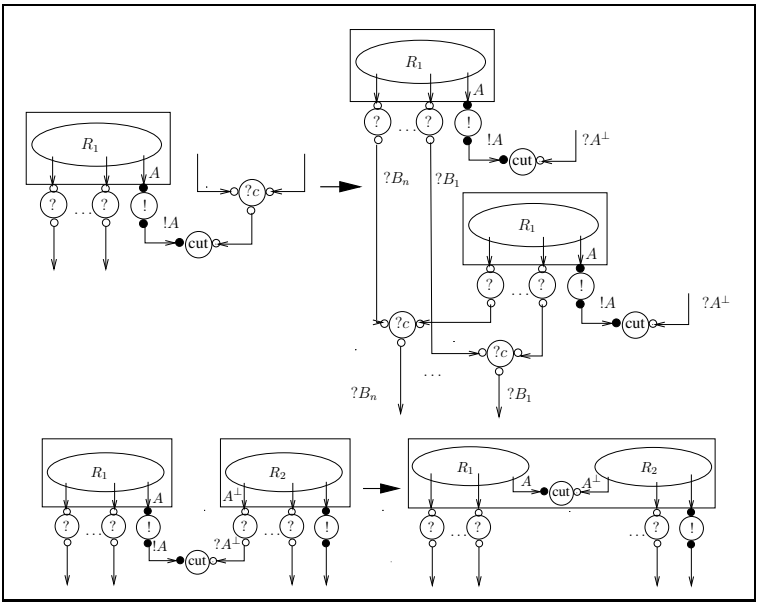


Fig. 5. Reduction steps (2/3)

Let us denote by R^i the proof-net at the beginning of round i. In order to bound the number of steps of this reduction strategy, the proof proceeds by, for $i = 0$ to $d(R)$:

- bounding the number of steps of round i by using $|R^i|_i$,
- bounding the size increase, that is to say bounding $|R^{i+1}|_{i+1}$ by using $|R^i|_i$.

Now, here we will essentially adapt the same reduction strategy and methodology for obtaining the bound, with the following modifications:

- on the strategy: we will not perform the reduction until obtaining a normal form (proof-net without cut), but we will stop when we can extract the result;
- on the methodology for obtaining the bound: we will use the assumption that the proof has a conclusion $!\mathbf{W} \multimap !^{k+2}\mathbf{B}$ and we will make a finer analysis of the size increase.

Now let us state the key Lemma that we will use:

Lemma 2 (Size bound). *Let R be a proof-net with:*

- *only exponential and weakening cuts at depth 0,*
- *k cuts at depth 0.*

Let R' be the proof-net obtained by reducing R at depth 0. Then we have:

$$|R'|_1 \leq |R|_0^k \cdot |R|_1.$$

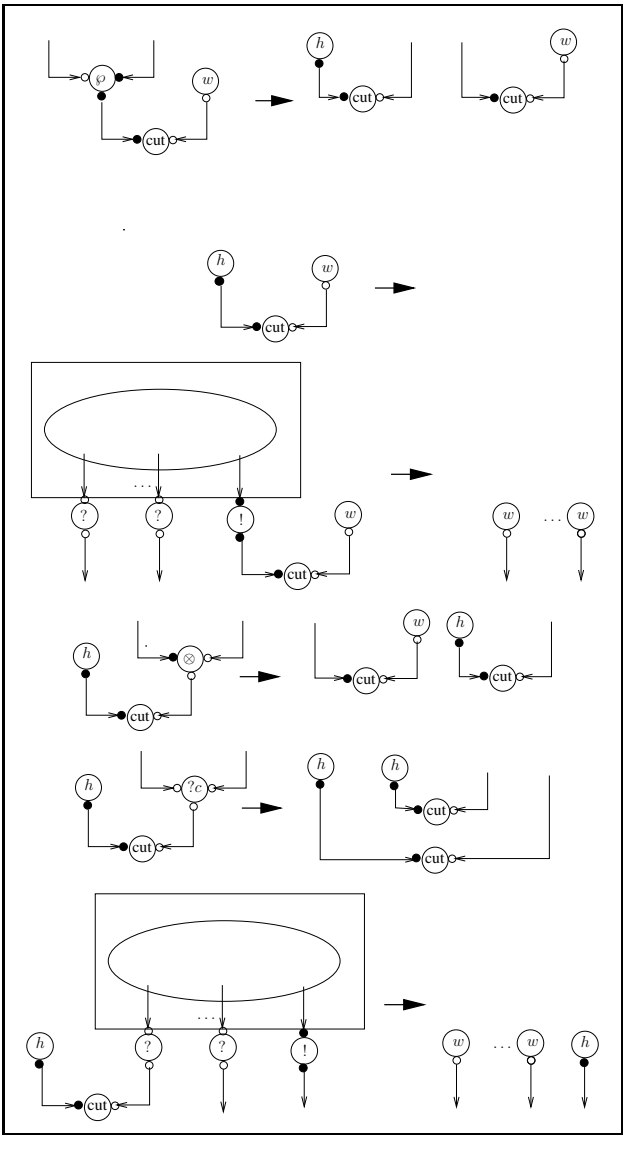


Fig. 6. Reduction steps: sample of weakening and h-steps (3/3)

So if we have a bound on the number k of cuts, we obtain a polynomial bound on the size of the proof-net R' after reduction at depth 0. In any case we can bound k by $|R|_0$, but then we basically recover the usual exponential bound [Gir98].

Proof. We say a contraction node of R at depth 0 is *active* if it is above a cut node. We denote by $|R|_a$ the number of active nodes of R at depth 0. Observe that we have $|R|_a + 1 \leq |R|_0$. We will prove the following statement by induction on k: $|R'|_1 \leq (|R|_a + 1)^k \cdot |R|_1$.

If $k = 0$ the result is trivial. Assume now the result valid for k and consider R with $k+1$ exponential cuts at depth 0.

First let us consider the case where there is a weakening cut among these $k+1$ cuts. We then reduce it persistently, following the weakening steps and h-steps of Fig. 6, and we end up with a proof-net R' such that: $|R'|_a \leq |R|_a$, $|R'|_1 \leq |R|_1$ and R' has k cuts at depth 0. We can then apply the induction hypothesis to R' and easily conclude.

Now, consider the case where we have $k+1$ exponential cuts. Using the correctness criterion one can check (e.g. as in [BP01]) that R admits a *special cut* c, that is to say an exponential cut for which the box $\mathcal{B}$ corresponding to the ! node does not have any cut below its auxilliary doors. We completely reduce the cut c, that is to say we reduce c and hereditarily all the cuts of its exponential tree until performing box-box or axiom reduction steps. The increase of size at depth 1 is due to the duplications of the box $\mathcal{B}$, which is copied at most $|R|_a$ times. Note that no active node is created during these reduction steps, because c is a special cut. We obtain in this way a proof-net R' such that: $|R'|_1 \leq (|R|_a + 1) \cdot |R|_1$, $|R'|_a \leq |R|_a$ and R' has k cuts at depth 0.

Besides, by induction hypothesis we have that R' can be reduced to R'' which is normal at depth 0 and: $|R''|_1 \leq (|R'|_a + 1)^k \cdot |R'|_1$. Combining the inequalities we thus get: $|R''|_1 \leq (|R|_a + 1)^{k+1} \cdot |R|_1$. We conclude by using the fact that $|R|_a + 1 \leq |R|_0$. □

We will need another result:

Lemma 3 (Readback). *Let R be a proof-net of conclusion* $\mathbf{B}$ *which only has exponential cuts at depth 0. Given R, one can in constant time decide whether it reduces to true or false.*

The proof of this Lemma takes advantage of the restricted exponentials rules of EAL_μ; it can be found in [Bai11].

Finally we get:

Proposition 1 (P soundness). *Let R be a normal proof-net of conclusion* $!\mathbf{W} \vdash !^2\mathbf{B}$. *Then there exists a polynomial P such that: any proof-net obtained by cutting R with a proof-net representing a word of length n can be evaluated in time bounded by $P(n)$.*

Proof. First, note that there exists a constant a such that for any n, for any binary word w of length n, w can be represented by a proof-net R_w of size $|R_w| \leq a \cdot n$.

Now, let us examine the structure of R at depth 0. If $?W^\perp$ is obtained by a weakening it is trivial. Otherwise there is an integer $k \geq 1$ and a proof-net S of conclusion $\vdash \mathbf{W}^\perp, \ldots, \mathbf{W}^\perp, !\mathbf{B}$ with k formulas $\mathbf{W}^\perp$ such that: R is obtained from S by applying a !-box and a certain number k' of contraction rules on $?\mathbf{W}^\perp$ formulas.

Now let R_w be a proof-net representing a word w, and let T be the proof-net obtained by cutting R with a box enclosing R_w. The proof-net T can be

reduced in at most $2k'$ steps (at depth 0) into a proof-net T' consisting in a box containing S cut with k copies of R_w. Therefore $|T'| \leq |R| + k \cdot |R_w|$.

Then, since $\mathbf{W} = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$, by applying k quantification reduction steps and $2k$ multiplicative reduction steps (at depth 1) we get a proof-net T'' with not cut at depth 0 and only exponential and weakening cuts at depth 1. Note that there are at most $3k$ exponential cuts at depth 1 and that $|T''| \leq |T'| \leq |R| + k \cdot |R_w|$.

Now by applying Lemma 2 to T'' at depth 1, we get that by reducing T'' at depth 1 we obtain $T^{(3)}$ with no cut at depths 0, 1 and such that $|T^{(3)}|_2 \leq |T''|_1^{3k} \cdot |T''|_2 \leq (|R| + k \cdot |R_w|)^{3k+1}$. The important point to notice here is that $(3k + 1)$ does not depend on n.

Finally we perform on $T^{(3)}$, at depth 2, reduction of all cuts but exponential cuts. These only make the size decrease, by Lemma 1, and so the number of steps is bounded by $|T^{(3)}|_2$. We obtain in this way a proof-net $T^{(4)}$ of conclusion $!^2B$, which: (i) does not have any cut at depths 0 and 1, (ii) only has exponential cuts at depth 2.

By Lemma 3, applied here at depth 2, the result can then be computed in constant time. So on the whole the computation has been carried out in a number of steps which is polynomial in $|R_w|$, hence polynomial in n.

Moreover, as the size of each intermediary proof-net in the reduction sequence is polynomially-bounded w.r.t. n, each reduction step can be performed in polynomial time (on a Turing machine), hence the whole reduction is performed in polynomial time. □

Proposition 2 (k-EXP soundness). *Let R be a normal proof-net of conclusion $!\mathbf{W} \vdash !^{k+2}\mathbf{B}$. Then there exists a polynomial P such that: any proof-net obtained by cutting R with a proof-net representing a word of length n can be evaluated in time bounded by $2_k^{P(n)}$.*

The proof of Prop. 2 is based on a generalization of the proof of Prop. 1, and needs an extra Lemma. See [Bai11] for more detail.

4 Extensional Completeness

To prove the extensional completeness results we will need another datatype using type fixpoints, that of *Scott binary words*:

$$\mathbf{W_S} = \mu\beta.\forall\alpha.(\beta \multimap \alpha) \multimap (\beta \multimap \alpha) \multimap (\alpha \multimap \alpha).$$

Scott words have already been used in several works on implicit complexity [DLB06, BT10, RV10]. One can easily define terms for the basic operations on binary words over the type $\mathbf{W_S}$:

$$\begin{array}{lll} cons_i & = \lambda w.\lambda s_0.\lambda s_1.\lambda x.(s_i\ w) & : \mathbf{W_S} \multimap \mathbf{W_S} \quad \text{for } i = 0, 1 \\ nil & = \lambda s_0.\lambda s_1.\lambda x.x & : \mathbf{W_S} \\ tail & = \lambda w.(w\ id\ id\ nil) & : \mathbf{W_S} \multimap \mathbf{W_S} \end{array}$$

where $id = \lambda x.x$.

For this datatype we can define a term for case distinction on words:

$$case \ : \ \forall\alpha.(\mathbf{W_S} \multimap \alpha) \multimap (\mathbf{W_S} \multimap \alpha) \multimap \alpha \multimap (\mathbf{W_S} \multimap \alpha)$$
$$case = \lambda F_0.\lambda F_1.\lambda a.\lambda w.(w \ F_0 \ F_1 \ a)$$

Remark 2. Actually Scott words can also be typed in elementary affine logic, without fixed points, e.g. with the following type: $\forall P.(P \multimap \forall X.((P \multimap X \multimap P) \multimap P) \multimap P)$. However it is not clear if one could define a *case* function in this setting.

We define the following type representing the configurations of a one-tape Turing machine over a binary alphabet, with n states:

$$\mathbf{Config} = \mathbf{W_S} \otimes \mathbf{B} \otimes \mathbf{W_S} \otimes \mathbf{B^n}$$

Given an element of this type: the first component represents the left part of the tape, in reverse order; the second component represents the symbol scanned by the head ; the third component represents the right part of the tape; the fourth part represents the current state.

Lemma 4. *Let q be a polynomial of $\mathbb{N}[X]$. We have:*

1. *there exists a proof of $!\mathbf{N} \multimap !\mathbf{N}$ representing the function $q(n)$;*
2. *for any $k \geq 1$, there exists a proof of $!\mathbf{N} \multimap !^{k+1}\mathbf{N}$ representing the function $2_k^{q(n)}$.*

Lemma 5. *Let $\mathcal{M}$ be a one-tape deterministic Turing machine. One can define:*

$$init : \mathbf{W_S} \multimap \mathbf{Config}, \ step : \mathbf{Config} \multimap \mathbf{Config}, \ accept? : \mathbf{Config} \multimap \mathbf{B},$$

such that:

- *given a binary word, init produces the corresponding initial configuration of the machine,*
- *the term step computes one step of the machine on a given configuration,*
- *given a configuration, the term accept? returns true (resp. false) if its state is accepting (resp. rejecting).*

The terms *init* and *accept?* are easy. The term *step* can be constructed based on the transition function of $\mathcal{M}$, by doing a case distinction, using the term *case*, as in [DLB06] (Sect.7, Lemma 4).

Remark 3. The configuration type of (Asperti Roversi 2002) in LAL does not use type fixpoints and can be directly adapted in EAL (replacing the § connective by !). Let us denote here this EAL type by $\mathbf{Config_C}$, as it is defined by using a Church encoding of binary words, while **Config** is defined by using the Scott encoding:

$$\mathbf{Config_C} = \forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha \multimap (\alpha \otimes \alpha \otimes \mathbf{B^n})),$$

where $\mathbf{B^n}$ is for representing the state of the machine.

Note that $\mathbf{Config_C}$ allows to define a function $step_C$ with the analogous type $\mathbf{Config_C} \multimap \mathbf{Config_C}$. However the corresponding term $accept?_C$ has type $\mathbf{Config_C} \multimap !\mathbf{B}$. It does not seem possible to give a term for this purpose of type $\mathbf{Config_C} \multimap \mathbf{B}$.

This is why we are considering here EAL_μ with type fixpoints, so as to be able to use Scott binary words.

Proposition 3 (Extensional completeness)

1. *Let f be a function representing a predicate of* P. *There exists a proof of conclusion* $!\mathbf{W} \vdash !^2\mathbf{B}$ *representing* f.
2. *Consider* $k \geq 1$ *and let f be a function representing a predicate of* $\mathsf{k\text{-}EXP}$. *There exists a proof of conclusion* $!\mathbf{W} \vdash !^{k+2}\mathbf{B}$ *representing* f.

The proof of this Proposition is based on simulations of Turing machines with the corresponding complexity bounds, using the results of Lemmas 4 and 5. See [Bai11] for more detail.

Finally, together the results of Propositions 1, 2 and 3 establish Theorem 1.

Remark 4. Observe that proofs of type $!\mathbf{W} \vdash !^2\mathbf{W}$ do not correspond to polynomial time functions. Indeed, one can easily define a proof *wdouble* of conclusion $\mathbf{W} \multimap \mathbf{W}$ which doubles the length of a word. By using iteration on words, applied here to this *wdouble* proof, one gets a proof *wexp* of conclusion $\mathbf{W} \vdash !\mathbf{W}$, which, given a word of length n, produces a word of length 2^n. Applying the ! rule one thus obtains a proof of $!\mathbf{W} \vdash !^2\mathbf{W}$ with the same behaviour.

To characterize the complexity class FP of polynomial time functions, one could use the type $!\mathbf{W} \multimap !^2\mathbf{W_S}$, where $\mathbf{W_S}$ is the type of Scott binary words. However a drawback of this characterization is that the proofs representing these functions could not be composed, because of the mismatch on the input and output types.

Remark 5. The proof of the previous theorem could be adapted in EAL (without fixpoint) instead of EAL_μ, by using the type $\mathbf{Config_C}$ of Remark 3, instead of $\mathbf{Config}$. But as we have the type $accept?_C : \mathbf{Config_C} \multimap !\mathbf{B}$, we would thus get in the end the type $!\mathbf{W} \multimap !^3\mathbf{B}$ for the simulation of a polynomial time machine. This in turn would not provide a P soundness result.

5 Conclusion and Future Work

Elementary linear logic was up to now considered as a simple variant of elementary linear logic with good structural properties but of limited interest for complexity. We have shown here that, provided one adds to it type fixpoints, it is expressive enough to characterize P, EXP and a time hierarchy inside the

elementary class. An interesting feature is that this provides a single type system in which one characterizes different complexity classes with the same term calculus, simply by considering terms of different types.

Several questions remain open. Is it possible to obtain the same result without type fixpoint? Could one also characterize in this system PSPACE and other space complexity classes in a way similar to [Lei02]? It would also be interesting to examine whether one could re-prove in this purely logical framework some classical hierarchy results, like $\mathsf{P} \neq \mathsf{EXP}$, by carrying out a diagonalisation argument.

Acknowledgements. The author would like to thank Jean-Yves Girard whose initial question, whether P could be characterized in elementary linear logic, triggered this work. Thanks also to Christophe Raffalli for useful discussions about the typing of Scott integers in system F.

This work was partially supported by project ANR-08-BLANC-0211-01 "COMPLICE".

References

[ACM04] Asperti, A., Coppola, P., Martini, S.: (Optimal) duplication is not elementary recursive. Information and Computation 193, 21–56 (2004)

[AR02] Asperti, A., Roversi, L.: Intuitionistic light affine logic. ACM Transactions on Computational Logic 3(1), 1–39 (2002)

[Bai11] Baillot, P.: Elementary linear logic revisited for polynomial time and an exponential time hierarchy (extended version). Technical Report, hal-00624742 (September 2011)

[BC92] Bellantoni, S., Cook, S.: New recursion-theoretic characterization of the polytime functions. Computational Complexity 2, 97–110 (1992)

[BGM10] Baillot, P., Gaboardi, M., Mogbil, V.: A Polytime Functional Language from Light Linear Logic. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 104–124. Springer, Heidelberg (2010)

[BMM11] Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations: a way to control resources. Theor. Comput. Sci. 412(25), 2776–2796 (2011)

[BP01] Baillot, P., Pedicini, M.: Elementary complexity and geometry of interaction. Fundamenta Informaticae 45(1-2), 1–31 (2001)

[BT09] Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. Inf. Comput. 207(1), 41–62 (2009); (A Preliminary Conference version appeared in the Proceedings of LICS 2004)

[BT10] Brunel, A., Terui, K.: Church => Scott = Ptime: an application of resource sensitive realizability. In: Proceedings International Workshop on Developments in Implicit Computational complExity (DICE 2010). EPTCS, vol. 23, pp. 31–46 (2010)

[DJ03] Danos, V., Joinet, J.-B.: Linear logic & elementary time. Information and Computation 183, 123–137 (2003)

[DLB06] Dal Lago, U., Baillot, P.: Light affine logic, uniform encodings and polynomial time. Mathematical Structures in Computer Science 16(4), 713–733 (2006)

[Gir87] Girard, J.-Y.: Linear logic. Theoretical Computer Science 50(1), 1–102 (1987)

[Gir98] Girard, J.-Y.: Light linear logic. Information and Computation 143, 175–204 (1998)

[GMRDR08] Gaboardi, M., Marion, J.-Y., Ronchi Della Rocca, S.: A logical account of Pspace. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008). ACM (2008)

[GR07] Gaboardi, M., Ronchi Della Rocca, S.: A Soft Type Assignment System for Lambda- Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 253–267. Springer, Heidelberg (2007)

[HAH11] Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proceedings of Symposium on Principles of Programming Languages (POPL 2011), pp. 357–370. ACM (2011)

[HJ06] Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)

[Hof03] Hofmann, M.: Linear types and non-size-increasing polynomial time computation. Inf. Comput. 183(1), 57–85 (2003)

[Jon01] Jones, N.D.: The expressive power of higher-order types or, life without cons. J. Funct. Program. 11(1), 5–94 (2001)

[Laf04] Lafont, Y.: Soft linear logic and polynomial time. Theoret. Comput. Sci. 318(1-2), 163–180 (2004)

[Lam96] Lamarche, F.: From proof nets to games. Electr. Notes Theor. Comput. Sci. 3, 107–119 (1996)

[Lei91] Leivant, D.: A foundational delineation of computational feasiblity. In: Proceedings Symposium on Logic in Computer Science (LICS 1991), pp. 2–11. IEEE Computer Society (1991)

[Lei94] Leivant, D.: Predicative recurrence and computational complexity I: word recurrence and poly-time. In: Feasible Mathematics II, pp. 320–343. Birkhäuser (1994)

[Lei02] Leivant, D.: Calibrating computational feasibility by abstraction rank. In: Proceedings LICS 2002, pp. 345–353. IEEE Computer Society (2002)

[RV10] Roversi, L., Vercelli, L.: Safe Recursion on Notation into a Light Logic by Levels. In: Proceedings of International Workshop on Developments in Implicit Computational complexity (DICE 2010). EPTCS, vol. 23, pp. 63–77 (2010)

A Proof Pearl with the Fan Theorem and Bar Induction

Walking through Infinite Trees with Mixed Induction and Coinduction

Keiko Nakata[1], Tarmo Uustalu[1], and Marc Bezem[2]

[1] Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, 12618 Tallinn, Estonia
{keiko,tarmo}@cs.ioc.ee
[2] Institutt for Informatikk, Universitet i Bergen,
Postboks 7800, 5020 Bergen, Norway
bezem@ii.uib.no

Abstract. We study temporal properties over infinite binary red-blue trees in the setting of constructive type theory. We consider several familiar path-based properties, typical to linear-time and branching-time temporal logics like LTL and CTL*, and the corresponding tree-based properties, in the spirit of the modal μ-calculus. We conduct a systematic study of the relationships of the path-based and tree-based versions of "eventually always blueness" and mixed inductive-coinductive "almost always blueness" and arrive at a diagram relating these properties to each other in terms of implications that hold either unconditionally or under specific assumptions (Weak Continuity for Numbers, the Fan Theorem, Lesser Principle of Omniscience, Bar Induction).

We have fully formalized our development with the Coq proof assistant.

1 Introduction

In this paper, we study temporal properties over infinite binary red-blue trees in the setting of constructive type theory. We consider several familiar path-based properties, typical to linear-time and branching-time temporal logics like LTL and CTL*, and the corresponding tree-based properties, in the spirit of the modal μ-calculus. Classically, some of these properties coincide, but in our more discerning setting they come out generally inequivalent. We then look for weak assumptions under which they imply each other. It turns out that some implications are in fact equivalent to principles well known in constructive mathematics and others follow from such principles.

We are primarily interested in path-based and tree-based variations of the properties of "eventually always blueness" and "almost always blueness" of a tree where the latter is defined by mixing induction and coinduction. We conduct a systematic study of the relationships of these properties and arrive at a diagram

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 353–368, 2011.

where we describe how these properties relate to each other in terms of implications that hold either unconditionally or under specific assumptions (Weak Continuity for Numbers, the Fan Theorem, Lesser Principle of Omniscience, Bar Induction). This way, we learn about the relative constructive strength of these properties in terms of the computational content of the assumptions used (cf., [11,12]) and conversely we get some intuition about the significance of these principles from a programmer's viewpoint.

The paper proceeds as follows. After setting up the basic framework in Sect. 2, we first study universal properties over paths in a tree and compare the path-based and tree-based variations. In Sect. 3.1, we examine trees that are always blue and, in Sect. 3.2, we also look at trees that are eventually red. In Sect. 4.1, we study eventually always blue trees. In Sect. 4.2, we consider trees that are almost always blue in the sense of a mixed inductive-coinductive definition.

We then continue with always eventually and infinitely often red trees (Sect. 5). Our journey ends with a short discussion of existential properties over paths in a tree (Sect. 6). We discuss related work in Sect. 7 to conclude in Sect. 8.

In regards to the various non-constructive principles we employ we use the terminology of Troelstra and van Dalen [22]. We use Latin lower case letters to represent finite objects, and Greek lower case letters for infinite objects. We present the definitions of both inductive and coinductive types and predicates in terms of sets of rules. The rules of inductive definitions are denoted by a single line and the rules of coinductive definitions are marked by a double line.

We have fully formalized our development in Coq. The Coq development is available at `http://cs.ioc.ee/~keiko/cypress.tgz`.

2 Preliminaries

In this section, we set up a basis for our development in the paper.

We have two colors, red (R) and blue (B). We also have steps, represented by bits, 0 for left and 1 for right. Namely,

$$\frac{}{R : \mathit{color}} \quad \frac{}{B : \mathit{color}} \qquad \frac{}{0 : \mathit{step}} \quad \frac{}{1 : \mathit{step}}$$

Streams $\alpha : A^\omega$ over a set A are infinite sequences over A defined coinductively by

$$\frac{a : A \quad \alpha : A^\omega}{a\ \alpha : A^\omega}$$

Bisimilarity on streams, $\alpha \sim \alpha'$, is also defined coinductively by

$$\frac{\alpha \sim \alpha'}{a\ \alpha \sim a\ \alpha'}$$

Our trees $\tau : \mathit{tree}$ are infinite binary trees with colored nodes, defined coinductively by

$$\frac{c : \mathit{color} \quad \tau_0 : \mathit{tree} \quad \tau_1 : \mathit{tree}}{\tau_0\ c\ \tau_1 : \mathit{tree}}$$

with bisimilarity on them, $\tau \sim \tau'$, defined coinductively by

$$\frac{\tau_0 \sim \tau_0' \quad \tau_1 \sim \tau_1'}{(\tau_0\ c\ \tau_1) \sim (\tau_0'\ c\ \tau_1')}$$

Note that a tree has no leaves, hence all the paths are infinite.

The relations $\sim$ (for both streams and trees) are straightforwardly seen to be equivalences. We take bisimilarity as the equality on streams (resp. trees), i.e., type-theoretically we treat streams (resp. trees) as a setoid with bisimilarity as the equivalence relation. Accordingly, we have to ensure that all functions and predicates we define on streams (resp. trees) are setoid functions and predicates (i.e., respect our notions of equality for them).

Lists $\ell : A^*$ over a set A are finite sequences over A defined inductively by

$$\frac{}{\langle\rangle : A^*} \qquad \frac{a : A \quad \ell : A^*}{a\ \ell : A^*}$$

The notation $\langle a\rangle$ denotes singletons, i.e., $\langle a\rangle = a\ \langle\rangle$. For $\ell, \ell' : A^*$, we denote by $\ell * \ell'$ the concatenation ℓ and ℓ'.

Concatenation can be extended to concatenation of a finite sequence $\ell : A^*$ and an infinite one $\alpha : A^\omega$:

$$\langle\rangle * \alpha = \alpha \qquad (a\ \ell) * \alpha = a\ (\ell * \alpha)$$

Non-empty lists $\ell : A^+$ over a set A are inductively defined by

$$\frac{a : A}{\langle a\rangle : A^+} \qquad \frac{a : A \quad \ell : A^+}{a\ \ell : A^+}$$

When necessary, we tacitly coerce non-empty lists to lists.

The function $\mathit{flatten} : (A^+)^\omega \to A^\omega$ flattens the given stream of non-empty lists over A. Formally, we define it by mutual corecursion together with an auxiliary function *flattenseq*:

$$\mathit{flatten}\,((\langle a\rangle\ \alpha) = a\ (\mathit{flatten}\,\alpha) \qquad \mathit{flatten}\,((a\ \ell)\ \alpha) = a\ (\mathit{flattenseq}\,\alpha\,\ell)$$
$$\mathit{flattenseq}\,\alpha\,\langle a\rangle = a\ (\mathit{flatten}\,\alpha) \qquad \mathit{flattenseq}\,\alpha\,(a\ \ell) = a\ (\mathit{flattenseq}\,\alpha\,\ell)$$

The initial segment of length n of a stream α is denoted by $\overline{\alpha}n$. Formally

$$\overline{\alpha}0 = \langle\rangle \qquad \overline{(a\ \alpha)}(n+1) = a\ (\overline{\alpha}n)$$

The suffix of a stream α at n, $\alpha@n$, is defined by

$$\alpha@0 = \alpha \quad (a\ \alpha)@(n+1) = \alpha@n$$

The subtree of a tree $\tau : \mathit{tree}$ at a *position* $p : \mathit{step}^*$ is denoted by $\tau@p$. Formally,

$$\tau@\langle\rangle = \tau \quad (\tau_0\ c\ \tau_1)@(0\ p) = \tau_0@p \quad (\tau_0\ c\ \tau_1)@(1\ p) = \tau_1@p$$

For a tree $\tau : \mathit{tree}$ and a *path* $\pi : \mathit{step}^\omega$, $[\![\tau]\!]_\pi$ returns the stream of colors in τ along π. Formally,

$$[\![\tau_0\ c\ \tau_1]\!]_{0\ \pi} = c\ [\![\tau_0]\!]_\pi \qquad [\![\tau_0\ c\ \tau_1]\!]_{1\ \pi} = c\ [\![\tau_1]\!]_\pi$$

Analogously, for a position $p : \mathit{step}^*$, $[\![\tau]\!]_p$ returns the list of colors in τ along p. Formally,

$$[\![\tau]\!]_{\langle\rangle} = \langle\rangle \qquad [\![\tau_0\ c\ \tau_1]\!]_{(0\ p)} = c\ [\![\tau_0]\!]_p \qquad [\![\tau_0\ c\ \tau_1]\!]_{(1\ p)} = c\ [\![\tau_1]\!]_p$$

It is convenient to introduce some predicates on streams of colors, typically written $\sigma : \mathit{color}^\omega$, and trees as primitives into our language for them. We define

$$\overline{\mathit{red}\ (R\ \sigma)} \quad \overline{\mathit{blue}\ (B\ \sigma)} \qquad \overline{\mathit{red}\ (\tau_0\ R\ \tau_1)} \quad \overline{\mathit{blue}\ (\tau_0\ B\ \tau_1)}$$

For streams of colors, we also define

$$\frac{X\ \sigma}{\mathcal{F}\ X\ \sigma} \quad \frac{\mathcal{F}\ X\ \sigma}{\mathcal{F}\ X\ (c\ \sigma)} \qquad\qquad \frac{X\ (c\ \sigma) \quad \mathcal{G}\ X\ \sigma}{\mathcal{G}\ X\ (c\ \sigma)}$$

Here, $\mathcal{F}$ and $\mathcal{G}$ are the "sometime in the future" ("finally") and "always in the future" ("globally") modalities of linear-time temporal logic. They are predicates on streams of colors parameterized over predicates X on streams of colors.[1] Analogously, we define "eventually" and "always" predicates on trees:

$$\frac{X\ \tau}{\mathcal{F}\ X\ \tau} \quad \frac{\mathcal{F}\ X\ \tau_0 \quad \mathcal{F}\ X\ \tau_1}{\mathcal{F}\ X\ (\tau_0\ c\ \tau_1)} \qquad\qquad \frac{X\ (\tau_0\ c\ \tau_1) \quad \mathcal{G}\ X\ \tau_0 \quad \mathcal{G}\ X\ \tau_1}{\mathcal{G}\ X\ (\tau_1\ c\ \tau_1)}$$

Again, $\mathcal{F}$ and $\mathcal{G}$ are predicates on trees parameterized over predicates X on trees. In section 6, we will consider variations of $\mathcal{G}\ X\ \tau$ and $\mathcal{F}\ X\ \tau$ which pick up one of the subtrees at every node as they go down through τ.

3 Always Blue and Eventually Red Trees

3.1 Always Blue Trees

A stream of colors $\sigma : \mathit{color}^\omega$ is *always blue*, if σ is "globally" blue, or $\mathcal{G}\ \mathit{blue}\ \sigma$. Similarly, a tree $\tau : \mathit{tree}$ is *always blue*, if every node of τ is blue, or $\mathcal{G}\ \mathit{blue}\ \tau$.

A tree is always blue if and only if every path of the tree is always blue:

Proposition 1. $\forall \tau : \mathit{tree}.\ \mathcal{G}\ \mathit{blue}\ \tau \Leftrightarrow (\forall \pi : \mathit{step}^\omega.\ \mathcal{G}\ \mathit{blue}\ [\![\tau]\!]_\pi)$.

[1] There is no need to see them as "first-class" predicate transformers, as there is no real impredicativity involved: the argument of $\mathcal{F}$ is constantly X in the definition of $\mathcal{F}$, and the same is true of the definition of $\mathcal{G}$.

3.2 Eventually Red Trees

A stream of colors σ is *eventually red* if σ is red at some position, or, $\mathcal{F}\,\mathit{red}\,\sigma$. An infinite tree τ is *eventually red* if a finite initial fragment of it has all leaves red, or $\mathcal{F}\,\mathit{red}\,\tau$.

Constructively, we have neither that a stream of colors is either always blue or eventually red, $\forall\sigma.\,\mathcal{G}\,\mathit{blue}\,\sigma \vee \mathcal{F}\,\mathit{red}\,\sigma$, nor that a stream of colors not being always blue implies that it is eventually red, $\forall\sigma.\,\neg\mathcal{G}\,\mathit{blue}\,\sigma \Rightarrow \mathcal{F}\,\mathit{red}\,\sigma$. The former is equivalent to the Lesser Principle of Omniscience (LPO), saying that $(\forall n.\,P\,n \vee \neg P\,n) \Rightarrow \forall n.\,\neg P\,n \vee \exists n.\,P\,n$, the latter to Markov's Principle (MP), saying that $(\forall n.\,P\,n \vee \neg P\,n) \Rightarrow \neg\forall n.\,\neg P\,n \Rightarrow \exists n.\,P\,n$ where P is a predicate on natural numbers. Both LPO and MP are important principles that are neither valid nor inconsistent constructively, but are valid classically. LPO is a special case of the Principle of the Excluded Middle (PEM). MP, which is a special case of the Double Negation Elimination, is even computationally meaningful, being realizable by search that we know cannot diverge.

If a tree is eventually red, then every path of the tree is eventually red:

Proposition 2. $\forall\tau : \mathit{tree}.\,\mathcal{F}\,\mathit{red}\,\tau \Rightarrow \forall\pi : \mathit{step}^\omega.\,\mathcal{F}\,\mathit{red}\,[\![\tau]\!]_\pi$.

To obtain the tree-based formulation from the path-based one, we invoke the Fan Theorem for a decidable bar ($\mathrm{FAN_D}$). Let P and Q be predicates on positions. Then $\mathrm{FAN_D}$ can be expressed as

$$(\forall p.\,P\,p \vee \neg P\,p) \Rightarrow \mathrm{FAN}$$

where FAN (the general Fan Theorem) is

$$(\forall\pi.\,\exists n.\,P\,(\overline{\pi}n)) \Rightarrow (\forall p.\,P\,p \Rightarrow Q\,p) \Rightarrow (\forall p.\,Q\,(p{*}\langle 0\rangle) \Rightarrow Q\,(p{*}\langle 1\rangle) \Rightarrow Q\,p) \Rightarrow Q\,\langle\rangle$$

$\mathrm{FAN_D}$ is not valid in basic constructive logic. It is the classical contrapositive of Weak König's Lemma[2], which is valid classically. In fact, Weak König's Lemma implies $\mathrm{FAN_D}$ even constructively [14].

$\mathrm{FAN_D}$ is both sufficient and necessary for path-based eventual redness to imply tree-based eventual redness.

Proposition 3. $\mathit{FAN}_D \Leftrightarrow (\forall\tau : \mathit{tree}.\,(\forall\pi : \mathit{step}^\omega.\,\mathcal{F}\,\mathit{red}\,[\![\tau]\!]_\pi) \Rightarrow \mathcal{F}\,\mathit{red}\,\tau)$.

Proof. $\Rightarrow$: The claim is an instance of $\mathrm{FAN_D}$ by taking P and Q as follows. For any $p : \mathit{step}^*$, $P\,p$ holds if the subtree of τ at p is red, or $\mathit{red}\,(\tau@p)$. For any $p : \mathit{step}^*$, $Q\,p$ holds if the subtree of τ at p is eventually red, or $\mathcal{F}\,\mathit{red}\,(\tau@p)$.

$\Leftarrow$: We define a tree τ_P by corecursion such that $\mathit{red}\,(\tau_P@p)$ if and only if $P\,p$. Then the assumption $\forall\pi.\,\exists n.\,P\,(\overline{\pi}n)$ is equivalent to $\forall\pi.\,\mathcal{F}\,\mathit{red}\,[\![\tau_P]\!]_\pi$. The assumption $\forall\tau.\,(\forall\pi.\,\mathcal{F}\,\mathit{red}\,[\![\tau]\!]_\pi) \Rightarrow \mathcal{F}\,\mathit{red}\,\tau$ therefore gives us $\mathcal{F}\,\mathit{red}\,\tau_P$. Now $Q\,\langle\rangle$ follows from $\forall p.\,\mathcal{F}\,\mathit{red}\,(\tau_P@p) \Rightarrow Q\,p$ proved by induction on the proof of $\mathcal{F}\,\mathit{red}\,(\tau_P@p)$ using $\forall p.\,P\,p \Rightarrow Q\,p$ and $\forall p.\,Q\,(p * \langle 0\rangle) \Rightarrow Q\,(p * \langle 1\rangle) \Rightarrow Q\,p$.

[2] Weak König's Lemma states that every infinite binary tree has an infinite path.

4 Eventually Always vs. Almost Always Blue Trees

In this section we look at path-based and tree-based concepts of eventually always and almost always blue trees.

4.1 Eventually Always Blue Trees

A stream of colors σ is *eventually always blue*, if, from some position on, σ is always blue, or $\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,\sigma$. A tree τ is *eventually always blue* if all nodes beyond some finite initial fragment of it are blue, or $\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,\tau$.

Again, the tree-based formulation is stronger than the path-based one:

Proposition 4. $\forall\tau : \mathit{tree}.\,\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,\tau \Rightarrow \forall\pi : \mathit{step}^\omega.\,\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,[\![\tau]\!]_\pi$.

To obtain the tree-based formulation from the path-based one, we invoke Weak Continuity for Numbers (WC-N) and the general Fan Theorem (FAN). Let P be a predicate on pairs of a path and natural number. Then WC-N can be expressed as

$$(\forall\pi.\,\exists n.\,P(\pi,n)) \Rightarrow \forall\pi.\,\exists m.\,\exists n.\,\forall\pi'.\,\overline{\pi}m = \overline{\pi'}m \Rightarrow P(\pi',n)$$

While FAN is valid classically, WC-N contradicts classical logic, but is nonetheless consistent with basic constructive logic.

We derive the tree-based formulation from the path-based one in two steps, to highlight the use of each of the two principles separately. We therefore introduce an intermediate step that is half path-based, half tree-based.

For any given path π, a tree τ is *eventually always blue along* π if the subtree of τ at some point along π is all blue, or $\exists n.\,\mathcal{G}\;\mathit{blue}\,(\tau@\overline{\pi}n)$.

If we accept WC-N, then we have that if every path of a tree is eventually always blue, then the tree is eventually always blue along every path:

Proposition 5. *Assuming WC-N,* $\forall\tau : \mathit{tree}.\,(\forall\pi : \mathit{step}^\omega.\,\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,[\![\tau]\!]_\pi) \Rightarrow \forall\pi : \mathit{step}^\omega.\,\exists n.\,\mathcal{G}\;\mathit{blue}\,(\tau@\overline{\pi}n)$.

Proof. For any given τ, we suppose that, $\forall\pi.\,\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,[\![\tau]\!]_\pi$. By WC-N, we have that, $\forall\pi.\,\exists m.\,\exists n.\,\forall\pi'.\,\overline{\pi}m = \overline{\pi'}m \Rightarrow \mathcal{G}\;\mathit{blue}\,([\![\tau]\!]_{\pi'}@n)$, by taking $P(\pi,n)$ to mean $\mathcal{G}\;\mathit{blue}\,([\![\tau]\!]_\pi@n)$. This gives us that, $\forall\pi.\,\exists n.\,\forall\pi'.\,\mathcal{G}\;\mathit{blue}\,[\![\tau@\overline{\pi}n]\!]_{\pi'}$. We conclude that $\forall\pi.\,\exists n.\,\mathcal{G}\;\mathit{blue}\,(\tau@\overline{\pi}n)$ by Prop. 1, as required.

If we accept FAN, then we have that if a tree is eventually always blue along every path, then the tree is eventually always blue:

Proposition 6. *Assuming FAN,* $\forall\tau : \mathit{tree}.\,(\forall\pi : \mathit{step}^\omega.\,\exists n.\,\mathcal{G}\;\mathit{blue}\,(\tau@\overline{\pi}n)) \Rightarrow \mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,\tau$.

Proof. The claim is an instance of FAN by taking P and Q as follows. For any $p : \mathit{step}^*$, $P\,p$ holds if $\mathcal{G}\;\mathit{blue}\,(\tau@p)$. For any $p : \mathit{step}^*$, $Q\,p$ holds if $\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,(\tau@p)$.

With the above two propositions, we derive the tree-based formulation from the path-based one:

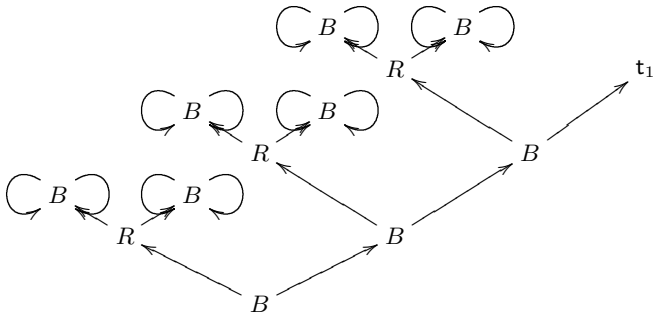


Fig. 1. The tree $\mathsf{t}_1 = (\mathsf{t}_0\ R\ \mathsf{t}_0)\ B\ \mathsf{t}_1$

Corollary 1. *Assuming WC-N and FAN,* $\forall \tau : \mathit{tree}.\, (\forall \pi : \mathit{step}^\omega.\, \mathcal{F}\,(\mathcal{G}\ \mathit{blue})\ [\![\tau]\!]_\pi) \Rightarrow \mathcal{F}\,(\mathcal{G}\ \mathit{blue})\ \tau.$

The concepts introduced are well illustrated by the following example.

Let t_0 be an always blue tree, defined by corecursion by

$$\mathsf{t}_0 = \mathsf{t}_0\ B\ \mathsf{t}_0$$

(this is in fact the only always blue tree, up to bisimilarity).

Our example of interest, t_1, is defined by corecursion by

$$\mathsf{t}_1 = (\mathsf{t}_0\ R\ \mathsf{t}_0)\ B\ \mathsf{t}_1$$

so that t_1 is red exactly at positions of the form 1^*0, i.e., it is red the first time a 0-step is taken. The tree is depicted in Fig. 1.

It is clear that $\mathcal{F}\,(\mathcal{G}\ \mathit{blue})\,\mathsf{t}_1$ is false, since it is impossible to carve out an finite initial fragment of t_1 such that the rest of the tree would be all blue. Similarly, $\forall \pi.\, \exists n.\, \mathcal{G}\ \mathit{blue}\ (\mathsf{t}_1 @ \overline{\pi} n)$ is false: the path 1^ω refutes it: there are red nodes beyond all positions on it.

At the same time $\forall \pi.\, \mathcal{F}\,(\mathcal{G}\ \mathit{blue})\ [\![\mathsf{t}_1]\!]_\pi$ is neither true nor false in basic constructive logic. Its truth is equivalent to every path either containing a turn to the left or always going to the right, which is LPO. With WC-N, however, one can conclude that the formula is false: this follows, e.g., from Prop. 5 and falsity of $\forall \pi.\, \exists n.\, \mathcal{G}\ \mathit{blue}\ (\mathsf{t}_1 @ \overline{\pi} n)$.

4.2 Almost Always Blue Trees

We proceed to two concepts of almost always blue trees. We obtain them by mixing induction and coinduction, more precisely, by nesting coinduction into induction in the style of [18].

We start with streams of colors that are almost always blue. They are defined as the least fixed point of a weak until operator in linear-time temporal logic. An equivalent definition is also found in the thesis of C. Raffalli [19]. The weak

until operator, $\mathcal{W}\,X$, is parameterized over any predicate X on streams of colors and is defined coinductively by

$$\frac{\mathcal{W}\,X\,\sigma}{\mathcal{W}\,X\,(B\;\sigma)} \qquad \frac{X\,\sigma}{\mathcal{W}\,X\,(R\;\sigma)}$$

so that $\mathcal{W}\,X\,\sigma$ holds if, whenever the first occurrence of red in σ is encountered, X holds on the suffix after the occurrence. Classically it is equivalent to that σ is either always blue or it is eventually red and X holds on the suffix after the first occurrence of red (which is guaranteed to exist as σ is eventually red). Our definition of $\mathcal{W}\,X$ avoids upfront decisions of LPO, i.e., whether the stream of colors is always blue or eventually red.

We then take the least fixed point of $\mathcal{W}\,X$. Define $\mu\mathcal{W}$ inductively in terms of $\mathcal{W}\,X$ by the (Park-style) rule:

$$\frac{\mathcal{W}\,\mu\mathcal{W}\,\sigma}{\mu\mathcal{W}\,\sigma}$$

As $\mathcal{W}\,X$ is monotone in X, the above definition makes sense. For the purpose of proof, in particular to avoid explicitly invoking monotonicity of the underlying predicate transformer $\mathcal{W}$, it is however convenient to use the Mendler-style rule:

$$\frac{\forall\sigma.\,X\,\sigma \Rightarrow \mu\mathcal{W}\,\sigma \quad \mathcal{W}\,X\,\sigma}{\mu\mathcal{W}\,\sigma}$$

The Park-style rule is derivable from the Mendler-style rule. We can also recover the inversion principle for $\mu\mathcal{W}$, thanks to the monotonicity of $\mathcal{W}\,X$ in X. We use the Mendler-style rule in our Coq formalization, as Coq's guardedness condition for coinduction nested into induction (as well as induction nested into coinduction) is often too weak to work with the Park style. The Mendler-style rule however requires impredicativity.

The statement $\mu\mathcal{W}\,\sigma$ does not give a clue as to where to find the red positions in σ or how many they are. Nonetheless it refutes that the stream of colors is infinitely often red (to be formulated below). We have previously scrutinized the definition of $\mu\mathcal{W}\,\sigma$, placed in a hierarchy of alternative definitions of streams of colors being finitely red, from the viewpoint of constructive mathematics [4]. In the remainder of the paper we refer to $\mu\mathcal{W}$ as mixed inductive-coinductive almost always blueness.

If a stream of colors is eventually always blue, then it is almost always blue:

Proposition 7. $\forall\sigma : \mathit{color}^{\omega}.\,\mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,\sigma \Rightarrow \mu\mathcal{W}\,\sigma$.

Analogously, we define trees that are almost always blue, $\mu\mathcal{W}\,\tau$, by taking the least fixed point of a weak-until operator for trees. This time, we only give the Park-style rule:

$$\frac{X\,\tau_0 \quad X\,\tau_1}{\mathcal{W}\,X\,(\tau_0\;R\;\tau_1)} \qquad \frac{\mathcal{W}\,X\,\tau_0 \quad \mathcal{W}\,X\,\tau_1}{\mathcal{W}\,X\,(\tau_0\;B\;\tau_1)} \qquad \frac{\mathcal{W}\,\mu\mathcal{W}\,\tau}{\mu\mathcal{W}\,\tau}$$

If a tree is eventually all blue, then it is almost always blue:

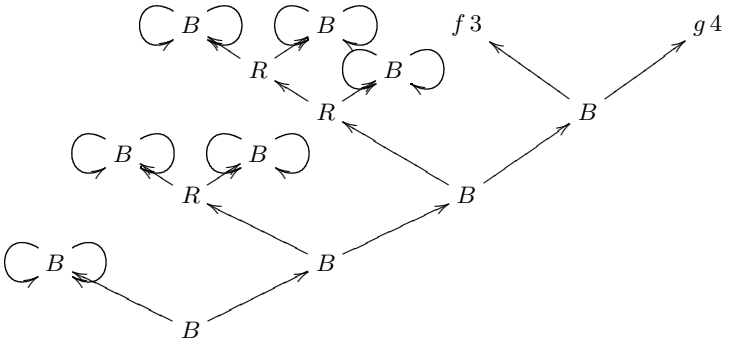


Fig. 2. The tree $\mathsf{t}'_1 = g\,0$, with the subtrees $f\,0$, $f\,1$, $f\,2$ fully expanded

Proposition 8. $\forall \tau : \mathit{tree}.\, \mathcal{F}\,(\mathcal{G}\;\mathit{blue})\,\tau \Rightarrow \mu\mathcal{W}\,\tau$.

Our example tree, t_1, is almost always blue.

Lemma 1. $\mu\mathcal{W}\,\mathsf{t}_1$.

Proof. We have $\mathcal{W}\,\mu\mathcal{W}\,\mathsf{t}_0$, proved by coinduction, therefore $\mu\mathcal{W}\,\mathsf{t}_0$, which yields $\mathcal{W}\,\mu\mathcal{W}\,(\mathsf{t}_0\ R\ \mathsf{t}_0)$.

We then prove $\mathcal{W}\,\mu\mathcal{W}\,\mathsf{t}_1$ by coinduction: we already know that $\mathcal{W}\,\mu\mathcal{W}\,(\mathsf{t}_0\ R\ \mathsf{t}_0)$ and by the coinduction hypothesis $\mathcal{W}\,\mu\mathcal{W}\,\mathsf{t}_1$, hence $\mu\mathcal{W}\,((\mathsf{t}_0\ R\ \mathsf{t}_0)\ B\ \mathsf{t}_1)$ as required.

To show another proof of almost always blueness, let us also consider a more reddish tree, t'_1, where the number of red nodes increases in proportion to the depth at which a 0-step is taken for the first time. The tree t'_1, depicted in figure 2, is defined as $g\,0$, where the functions $f, g : \mathit{nat} \to \mathit{tree}$ are defined by corecursion by

$$f\,0 = \mathsf{t}_0 \qquad f\,(n+1) = (f\,n)\ R\ \mathsf{t}_0 \qquad g\,n = (f\,n)\ B\ (g\,(n+1))$$

The tree t'_1 is almost always blue.

Lemma 2. $\mu\mathcal{W}\,\mathsf{t}'_1$.

Proof. We prove $\forall n.\, \mathcal{W}\,\mu\mathcal{W}\,(f\,n)$ by induction on n. We then prove $\forall n.\, \mathcal{W}\,\mu\mathcal{W}\,(g\,n)$ by coinduction, which yields $\mu\mathcal{W}\,(g\,0)$, namely $\mu\mathcal{W}\,\mathsf{t}'_1$, as required.

As usual, the tree-based formulation is stronger than the path-based one. We give the proof here to demonstrate the use of the Mendler-style induction.

Proposition 9. $\forall \tau : \mathit{tree}.\, \mu\mathcal{W}\,\tau \Rightarrow \forall \pi : \mathit{step}^\omega.\, \mu\mathcal{W}\,[\![\tau]\!]_\pi$.

Proof. By induction on the proof of $\mu\mathcal{W}\,\tau$. We are given as the induction hypothesis that, $\forall \tau.\, X\,\tau \Rightarrow \forall \pi.\, \mu\mathcal{W}\,[\![\tau]\!]_\pi$ for some predicate X on trees. We

also have $\mathcal{W}\, X\, \tau$. We have to prove that, $\forall \pi.\, \mu\mathcal{W}\, [\![\tau]\!]_\pi$. We do so by proving that, $\forall \tau.\, \mathcal{W}\, X\, \tau \Rightarrow \forall \pi.\, \mathcal{W}\, \mu\mathcal{W}\, [\![\tau]\!]_\pi$ by coinduction, using the main induction hypothesis.

In contrast to the earlier considered case of eventually always blue streams of colours, a proof that a stream of colors is almost always blue does not give us a position at which the suffix of the stream is all blue. Indeed, knowing it, i.e., $\forall \sigma.\, \mu\mathcal{W}\, \sigma \Rightarrow \mathcal{F}\, (\mathcal{G}\, \mathit{blue})\, \sigma$, is equivalent to LPO [4].

Lemma 3. $(\forall \sigma.\, \mathcal{F}\, \mathit{red}\, \sigma \vee \mathcal{G}\, \mathit{blue}\, \sigma) \Leftrightarrow (\forall \sigma.\, \mu\mathcal{W}\, \sigma \Rightarrow \mathcal{F}\, (\mathcal{G}\, \mathit{blue})\, \sigma)$.

Our proof to obtain the tree-based formulation from the path-based formulation is sketched as follows. We build infinitely branching trees from binary trees (the function $t2T$ defined below). We then find a decidable bar condition for these infinitely branching trees (Lemma 5). We know that, if every path of a binary tree τ is almost always blue, then a bar exists in the infinitely branching tree θ corresponding to τ (Cor. 2), therefore we can apply Bar Induction on θ (Lemma 7). This in turn proves that the original tree τ is almost always blue (Lemma 8). Below we will make this argument formal.

Our infinitely branching trees, $\theta : \mathit{Tree}$, have nodes labeled by binary trees and edges labeled by non-empty lists of steps. They are defined coinductively by

$$\frac{\tau : \mathit{tree} \quad f : \mathit{step}^+ \to \mathit{Tree}}{(\tau, f) : \mathit{Tree}}$$

A path in $\theta : \mathit{Tree}$ is characterized by a stream of non-empty lists of steps. For a tree $\theta : \mathit{Tree}$ and a position $q : (\mathit{step}^+)^*$, $\theta @ q : \mathit{Tree}$ and $[\![\theta]\!]_q : (\mathit{color}^+)^*$ are defined naturally by

$$(\tau, f)@\langle\rangle = (\tau, f) \qquad (\tau, f)@(p\ q) = (f\ p)@q$$
$$[\![(\tau, f)]\!]_{\langle\rangle} = \langle\rangle \qquad [\![(\tau, f)]\!]_{(p\ q)} = ([\![\tau]\!]_p)\ ([\![(f\ p)]\!]_q)$$

For $\rho : (\mathit{step}^+)^\omega$, $[\![\theta]\!]_\rho : (\mathit{color}^+)^\omega$ is defined analogously.

We define a function, $t2T : \mathit{tree} \to \mathit{Tree}$, from binary trees to infinitely branching trees by corecursion by

$$t2T\, \tau = (\tau,\ \lambda p : \mathit{step}^+.\, t2T\, \tau @ p)$$

so that, for any position $q : (\mathit{step}^+)^*$, the label of $t2T\, \tau$ at q is the subtree of τ at $\mathit{flatten}\, q$ (assuming $\mathit{flatten}$ is extended to finite sequences of non-empty lists in an obvious way). In particular, the streams of colors in $t2T\, \tau$ and τ along a path $\rho : (\mathit{step}^+)^\omega$ agree up to flattening. This is what the next lemma proves.

Lemma 4. $\forall \tau : \mathit{tree}, \rho : (\mathit{step}^+)^\omega.\, \mathit{flatten}\, [\![t2T\, \tau]\!]_\rho \sim [\![\tau]\!]_{(\mathit{flatten}\, \rho)}$.

A non-empty list of colors $s : \mathit{color}^+$ is good, $\mathit{good}\, s$, if s is of the form $B^* R$. Formally,

$$\frac{}{\mathit{good}\, \langle R \rangle} \qquad \frac{\mathit{good}\, s}{\mathit{good}\, (B\ s)}$$

The predicate good is decidable:

Lemma 5. $\forall s : color^+.\ good\ s \vee \neg good\ s.$

We will use $\neg good$ as the bar condition.

A stream over non-empty lists of colors $\alpha : (color^+)^\omega$, is wellfounded, $wf\ \alpha$, if α contains a color list that is not good. Formally,

$$\frac{\neg good\ s}{wf\ (s\ \alpha)} \qquad \frac{good\ s \quad wf\ \alpha}{wf\ (s\ \alpha)}$$

Then we have that, for any $\alpha : (color^+)^\omega$, if $flatten\ \alpha$ is almost always blue, then α is wellfounded:

Lemma 6. $\forall \alpha : (color^+)^\omega.\ \mu\mathcal{W}\ (flatten\ \alpha) \Rightarrow wf\ \alpha.$

As a corollary to Lemmata 4 and 6, we obtain that, if every path of a tree τ is almost always blue, then every path of $t2T\ \tau$ is wellfounded:

Corollary 2. $\forall \tau : tree.\ (\forall \pi : step^\omega.\ \mu\mathcal{W}\ [\![\tau]\!]_\pi) \Rightarrow \forall \rho : (step^+)^\omega.\ wf\ [\![(t2T\ \tau)]\!]_\rho.$

We lift wellfoundedness on streams of nonempty lists of colors to trees:

$$\frac{\forall p : step^+.\ good\ [\![\tau]\!]_p \Rightarrow wf\ (f\ p)}{wf\ (\tau, f)}$$

Now we are to apply Bar Induction (BI) (the generalization of $\mathrm{FAN_D}$ from binary trees to infinitely branching trees) to obtain wellfounded trees from trees whose paths are wellfounded. Let P and Q be predicates on lists of nonempty lists of steps. Noticing the isomorphism between natural numbers and nonempty lists of steps, Bar Induction can be expressed as

$$\begin{aligned}&(\forall q : (step^+)^*.\ P\ q \vee \neg P\ q) \Rightarrow (\forall \rho : (step^+)^\omega.\ \exists n.\ P\ (\overline{\rho} n)) \Rightarrow \\ &\quad (\forall q : (step^+)^*.\ P\ q \Rightarrow Q\ q) \Rightarrow (\forall q : (step^+)^*.\ (\forall p : step^+.\ Q\ (q * \langle p \rangle))) \Rightarrow Q\ q) \Rightarrow \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Q\ \langle\rangle\end{aligned}$$

If we accept BI, we have that, if every path of a tree $\theta : Tree$ is wellfounded, then θ is wellfounded:

Lemma 7. *Assuming BI,* $\forall \theta : Tree.\ (\forall \rho : (step^+)^\omega.\ wf\ [\![\theta]\!]_\rho) \Rightarrow wf\ \theta.$

Proof. The claim is an instance of BI by taking P and Q as follows. For any $q : (step^+)^*$, $P\ q$ holds if $[\![\theta]\!]_q = u * \langle s \rangle$ and $\neg good\ s$. For any $q : (step^+)^*$, $Q\ q$ holds if $P\ q$ or $wf\ (\theta @ q)$.

The following lemma says that, for any tree $\tau : tree$, if $t2T\ \tau$ is wellfounded then τ is almost always blue:

Lemma 8. $\forall \tau : tree.\ wf\ (t2T\ \tau) \Rightarrow \mu\mathcal{W}\ \tau.$

Finally, putting the above lemmata together, we have that if every path of a tree $\tau : tree$ is almost always blue, then τ is almost always blue:

Proposition 10. *Assuming BI,* $\forall \tau : tree.\ (\forall \pi : step^\omega.\ \mu\mathcal{W}\ [\![\tau]\!]_\pi) \Rightarrow \mu\mathcal{W}\ \tau.$

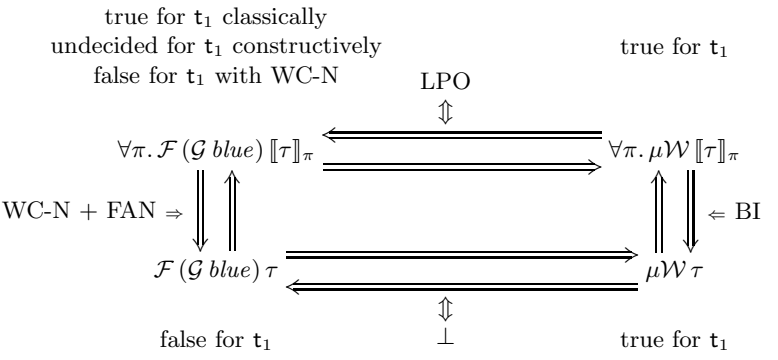


Fig. 3. Relations between the different concepts of almost always blue

For our example tree, we have $\mu\mathcal{W}\mathtt{t}_1$ but not $\mathcal{F}\,(\mathcal{G}\,\mathit{blue})\,\mathtt{t}_1$, hence tree-based almost always blueness does not imply tree-based eventually always blueness.

Proposition 11. $\exists\tau : \mathit{tree}.\,\mu\mathcal{W}\tau \wedge \neg\mathcal{F}\,(\mathcal{G}\,\mathit{blue})\,\tau$.

We can now take stock. The four properties $\forall\pi.\,\mathcal{F}\,(\mathcal{G}\,\mathit{blue})\,[\![\tau]\!]_\pi$, $\mathcal{F}\,(\mathcal{G}\,\mathit{blue})\,\tau$, $\forall\pi.\,\mu\mathcal{W}\,[\![\tau]\!]_\pi$, and $\mu\mathcal{W}\,\tau$ are interrelated as depicted in the diagram in Fig. 3. The implications that are annotated require additional assumptions as shown. The figure also displays the status of the example $\mathtt{t}_1$ wrt. each property. (The example $\mathtt{t}_1'$ has the same status as $\mathtt{t}_1$ in each case.)

5 Always Eventually Red vs. Infinitely Often Red Trees

We proceed to always eventually red and infinitely often red trees.

We define always eventually redness of a stream of colors σ as $\mathcal{G}\,(\mathcal{F}\,\mathit{red})\,\sigma$ (cf. [3, Ch. 13]). This definition is (classically) dual to the definition of streams of colors that are eventually always blue, $\mathcal{F}\,(\mathcal{G}\,\mathit{blue})\,\sigma$. The modalities $\mathcal{G}$ and $\mathcal{F}$ are flipped and so are the colors *red* and *blue*.

The definition of infinitely often redness of a stream of colors is obtained by dualizing the definitions of $\mathcal{W}\,X$ and $\mu\mathcal{W}$, yielding

$$\frac{\mathcal{U}\,X\,\sigma}{\mathcal{U}\,X\,(B\,\sigma)} \qquad \frac{X\,\sigma}{\mathcal{U}\,X\,(R\,\sigma)} \qquad \frac{\mathcal{U}\,\nu\mathcal{U}\,\sigma}{\nu\mathcal{U}\,\sigma}$$

The strong until operator $\mathcal{U}\,X$, parameterized over any predicate X on streams of colors, is dual to the weak until operator $\mathcal{W}\,X$: The statement $\mathcal{U}\,X\,\sigma$ says that the suffix of σ after the first occurrence of red must satisfy X and the occurrence must exist. Then $\nu\mathcal{U}$ takes the *greatest* fixed point of $\mathcal{U}\,X$, whereas $\mu\mathcal{W}$ was the least fixed point of $\mathcal{W}\,X$.

The two properties of streams of colors are equivalent:

Proposition 12. $\forall\sigma : \mathit{color}^\omega.\,\nu\mathcal{U}\,\sigma \Leftrightarrow \mathcal{G}\,(\mathcal{F}\,\mathit{red})\,\sigma$.

Classically, a stream of colors is almost always blue or infinitely often red:

Lemma 9. *Assuming PEM,* $\forall\sigma : \mathit{color}^\omega.\,\mu\mathcal{W}\,\sigma \vee \nu\mathcal{U}\,\sigma$.

Analogously, we define tree-based always eventually redness of τ as $\mathcal{G}\,(\mathcal{F}\,\mathit{blue})\,\tau$ and tree-based infinitely often redness as $\nu\mathcal{U}\,\tau$ defined by

$$\frac{\mathcal{U}\,X\,\tau_0 \quad \mathcal{U}\,X\,\tau_1}{\mathcal{U}\,X\,(\tau_0\ B\ \tau_1)} \qquad \frac{X\,\tau_0 \quad X\,\tau_1}{\mathcal{U}\,X\,(\tau_0\ R\ \tau_1)} \qquad \frac{\mathcal{U}\,\nu\mathcal{U}\,\tau}{\nu\mathcal{U}\,\tau}$$

Again, the two properties are equivalent:

Proposition 13. $\forall\tau : \mathit{tree}.\ \nu\mathcal{U}\,\tau \Leftrightarrow \mathcal{G}\,(\mathcal{F}\,\mathit{red})\,\tau.$

The tree-based property implies the path-based property:

Proposition 14. $\forall\tau : \mathit{tree}.\ \nu\mathcal{U}\,\tau \Rightarrow (\forall\pi : \mathit{step}^\omega.\ \nu\mathcal{U}\,[\![\tau]\!]_\pi).$

For the converse implication, we assume FAN_D:

Proposition 15. *Assuming* FAN_D, $\forall\tau : \mathit{tree}.\ (\forall\pi : \mathit{step}^\omega.\ \nu\mathcal{U}\,[\![\tau]\!]_\pi) \Rightarrow \nu\mathcal{U}\,\tau.$

6 Existential Properties

So far, we have been looking at universal properties over all paths of a tree. In this section, we turn them into existential properties. It turns out that the path-based and tree-based formulations are then necessarily equivalent.

We introduce two new primitives, $\mathcal{F}^\exists\,X$ and $\mathcal{G}^\exists\,X$, parameterized over tree predicates X, into our language for trees:

$$\frac{X\,\tau}{\mathcal{F}^\exists\,X\,\tau} \quad \frac{\mathcal{F}^\exists\,X\,\tau_0}{\mathcal{F}^\exists\,X\,(\tau_0\ c\ \tau_1)} \quad \frac{\mathcal{F}^\exists\,X\,\tau_1}{\mathcal{F}^\exists\,X\,(\tau_0\ c\ \tau_1)}$$

$$\frac{X\,(\tau_0\ c\ \tau_1) \quad \mathcal{G}^\exists\,X\,\tau_0}{\mathcal{G}^\exists\,X\,(\tau_0\ c\ \tau_1)} \qquad \frac{X\,(\tau_0\ c\ \tau_1) \quad \mathcal{G}^\exists\,X\,\tau_1}{\mathcal{G}^\exists\,X\,(\tau_0\ c\ \tau_1)}$$

In contrast to $\mathcal{F}\,X\,\tau$ and $\mathcal{G}\,X\,\tau$, the new primitives $\mathcal{F}^\exists\,X\,\tau$ and $\mathcal{G}^\exists\,X\,\tau$ step down through the tree, picking up one of the two subtrees at every node.

The path-based and tree-based properties that we have considered coincide, with the exception of "always eventually red", for which the path-based property is stronger. That the converse implication does not hold is witnessed by our example tree $\mathtt{t}_1$. The reason for the failure is that $\mathcal{G}^\exists\,(\mathcal{F}^\exists\,\mathit{red})\,\tau$ does not require the red nodes to be on the same path.

Proposition 16.
1. $\forall\tau : \mathit{tree}.\ (\exists\pi : \mathit{step}^\omega.\ \mathcal{G}\,\mathit{blue}\,[\![\tau]\!]_\pi) \Leftrightarrow \mathcal{G}^\exists\,\mathit{blue}\,\tau.$
2. $\forall\tau : \mathit{tree}.\ (\exists\pi : \mathit{step}^\omega.\ \mathcal{F}\,\mathit{red}\,[\![\tau]\!]_\pi) \Leftrightarrow \mathcal{F}^\exists\,\mathit{red}\,\tau.$
3. $\forall\tau : \mathit{tree}.\ (\exists\pi : \mathit{step}^\omega.\ \mathcal{F}\,(\mathcal{G}\,\mathit{blue})\,[\![\tau]\!]_\pi) \Leftrightarrow \mathcal{F}^\exists\,(\mathcal{G}^\exists\,\mathit{blue})\,\tau.$
4. $\forall\tau : \mathit{tree}.\ (\exists\pi : \mathit{step}^\omega.\ \mathcal{G}\,(\mathcal{F}\,\mathit{red})\,[\![\tau]\!]_\pi) \Rightarrow \mathcal{G}^\exists\,(\mathcal{F}^\exists\,\mathit{red})\,\tau.$
   $\exists\tau : \mathit{tree}.\ \mathcal{G}^\exists\,(\mathcal{F}^\exists\,\mathit{red})\,\tau \wedge \neg(\exists\pi : \mathit{step}^\omega.\ \mathcal{G}\,(\mathcal{F}\,\mathit{red})\,[\![\tau]\!]_\pi).$

For a tree having a path that is almost always blue or infinitely often red, we introduce corresponding weak until and strong until operators:

$$\frac{X\ \tau_0}{\mathcal{W}^\exists\, X\, (\tau_0\ R\ \tau_1)} \quad \frac{X\ \tau_1}{\mathcal{W}^\exists\, X\, (\tau_0\ R\ \tau_1)} \quad \frac{\mathcal{W}^\exists\, X\, \tau_0}{\mathcal{W}^\exists\, X\, (\tau_0\ B\ \tau_1)} \quad \frac{\mathcal{W}^\exists\, X\, \tau_1}{\mathcal{W}^\exists\, X\, (\tau_0\ B\ \tau_1)} \quad \frac{\mathcal{W}^\exists\, \mu\mathcal{W}^\exists\, \tau}{\mu\mathcal{W}^\exists\, \tau}$$

$$\frac{X\ \tau_0}{\mathcal{U}^\exists\, X\, (\tau_0\ R\ \tau_1)} \quad \frac{X\ \tau_1}{\mathcal{U}^\exists\, X\, (\tau_0\ R\ \tau_1)} \quad \frac{\mathcal{U}^\exists\, X\, \tau_0}{\mathcal{U}^\exists\, X\, (\tau_0\ B\ \tau_1)} \quad \frac{\mathcal{U}^\exists\, X\, \tau_1}{\mathcal{U}^\exists\, X\, (\tau_0\ B\ \tau_1)} \quad \frac{\mathcal{U}^\exists\, \nu\mathcal{U}^\exists\, \tau}{\nu\mathcal{U}^\exists\, \tau}$$

The path-based and tree-based properties are equivalent for both almost always blueness as well as infinitely often redness.

Proposition 17. $\forall\tau : \mathit{tree}.\, (\exists\pi : \mathit{step}^\omega.\, \mu\mathcal{W}\, [\![\tau]\!]_\pi) \Leftrightarrow \mu\mathcal{W}^\exists\tau.$

Proposition 18. $\forall\tau : \mathit{tree}.\, (\exists\pi : \mathit{step}^\omega.\, \nu\mathcal{U}\, [\![\tau]\!]_\pi) \Leftrightarrow \nu\mathcal{U}^\exists\, \tau.$

Lemma 10. *Assuming PEM,* $\forall\tau : \mathit{tree}.\, \mu\mathcal{W}\, \tau \vee \nu\mathcal{U}^\exists\, \tau$ *and* $\forall\tau : \mathit{tree}.\, \nu\mathcal{U}\, \tau \vee \mu\mathcal{W}^\exists\, \tau.$

7 Related Work

Dam [8] gave a direct translation from CTL* into the modal μ-calculus in a classical setting. Classically, the problem reduces to translation of formulae of the form $E\phi$ where ϕ is a linear-time formula, i.e., ϕ does not contain path quantifiers. Then the translation is given by carefully analyzing the tableau representing $E\phi$ and thereby characterizing infinite paths in the tableau by least or greatest fixpoints.

Formalizations of LTL, CTL* and the modal μ-calculus in Coq have been given by several authors (cf. [17,21,20,7,3]). These works study either LTL (or CTL*, which subsumes LTL) or the modal μ-calculus, and focus on different issues from ours, e.g. issues in encoding modal μ-calculus formulae in higher-order abstract syntax [17] or machine verification of a model checker for the modal μ-calculus [20]. Moreover, our use of mixed induction and coinduction for formalizing almost always bluemess and infinitely often redness appears new.

It is known that the Weak König's lemma, WKL, constructively implies FAN_D [13,14]. Moreover, a weakened from of WKL, which additionally requires that the tree under consideration has at most one infinite path, is equivalent to FAN_D [1]. A recent account of the computational content of the principles we use can be found in, e.g., [11,12] in that FAN is realized by the fan functional and bar induction is realized (in some sense) by bar recursion.

In our recent work [4] we studied alternative definitions of streams of colors being finitely red, including $\mathcal{F}\, (\mathcal{G}\, \mathit{blue})\, \sigma$ and $\mu\mathcal{W}\, \sigma$, and characterized their differences in strength in a precise way by weak instances of PEM. Coquand and Spiwack [6] introduced four notions of finiteness of sets in Bishop's set theory [5]. The two works exhibit a pleasant correspondence [4].

Mixed inductive-coinductive definitions seem to be quite fundamental in applications (e.g., infinitely often red, subtyping [9], the stream processors of

Hancock et al. [15], uniformly continuous functions on a compact real interval [2], weak bisimilarity and delay-free operational semantics of interactive programs [18]). Mendler-style (co)recursion [16] uses that a monotone (co)inductive definition is equivalent to a positive one, via a syntactic left (right) Kan extension along identity (e.g., instead of $\mu X.\, F\, X$ one works with $\mu X.\, \exists Y.\, (Y \to X) \times F\, Y$). We exploited this fact to enable Coq's structural recursion for an inductive definition with a nested coinductive definition and vice versa, at the price of impredicativity.

8 Conclusion

We analyzed several temporal operators from the point of view of constructive logic. We observed that, with operators like "eventually always" and "almost always", various classically equivalent definitions become inequivalent. Which one is more adequate in any actual application depends on the purpose at hand. It is also plausible that some of them have a smoother metatheory—more likely the tree-based ones, especially the tree-based "almost always".

We chose to treat streams and infinite trees as coinductive data, defined the temporal properties of interest in terms of inductive and coinductive predicates, and reasoned about them with induction and coinduction. We are pleased with the concision and elegance this approach offered, compared with more "low-level" arithmetized concepts as is more common in works on constructive mathematics.

We witnessed that the differences between the variations correspond to well-known principles from constructive mathematics, e.g., the implication from the path-based "eventually" operator to tree-based "eventually" is exactly the decidable Fan Theorem etc.

This demonstrates, to our mind, that the studies into constructive mathematics, which were initiated by Brouwer and elaborated by Bishop and others, and are not particularly well-known in the programming languages community, are not without significance for modern formalized programming theory or dependently typed programming.

In future work, we wish to reach a deeper understanding of the computational aspects in our results and their implications for programming and reasoning about interactive and concurrent systems.

Acknowledgments. We are indebted to Christine Paulin-Mohring, Hugo Herbelin, Thorsten Altenkirch for fruitful discussions.

K. Nakata and T. Uustalu's research was supported by ERDF through the Estonian Centre of Excellence in Computer Science (EXCS). M. Bezem's visit to Estonia in Feb. 2011 was supported by the same project.

References

1. Berger, J., Ishihara, H.: Brouwer's fan theorem and unique existence in constructive analysis. Math. Log. Quart. 51(4), 360–364 (2005)
2. Berger, U.: From coinductive proofs to exact real arithmetic: theory and applications. Logical Methods in Comput. Sci. 7(1) (2011)

3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
4. Bezem, M., Nakata, K., Uustalu, T.: On streams that are finitely red (submitted for publication 2011) (manuscript)
5. Bishop, E.: Foundations of Constructive Analysis. McGraw-Hill, New York (1967)
6. Coquand, T., Spiwack, A.: Constructively finite? In: Laureano Lambán, L., Romero, A., Rubio, J. (eds.) Scientific Contributions in Honor of Mirian Andrés Gómez. Universidad de La Rioja (2010)
7. Coupet-Grimal, S.: An axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. J. of Logic and Comput. 13(6), 801–813 (2003)
8. Dam, M.: CTL* and ECTL* as fragments of the modal mu-calculus. Theor. Comput. Sci. 126(1), 77–96 (1994)
9. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively: an exercise in mixed induction and coinduction. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 100–118. Springer, Heidelberg (2010)
10. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 905–1072. MIT Press (1990)
11. Escardó, M.H., Oliva, P.: Selection functions, bar recursion and backward induction. Math. Struct. in Comput. Sci. 20(2), 127–168 (2010)
12. Escardó, M.H., Oliva, P.: What sequential games, the Tychonoff Theorem and the double-negation shift have in common. In: Proc. of 3rd ACM SIGPLAN Wksh. on Mathematically Structured Functional Programming, MSFP 2010, pp. 21–32. ACM Press (2010)
13. Ishihara, H.: An omniscience principle, the König Lemma and the Hahn-Banach theorem. Math. Log. Quart. 36(3), 237–240 (1990)
14. Ishihara, H.: Weak König's lemma implies Brouwer's fan theorem: a direct proof. Notre Dame J. of Formal Logic 47(2), 249–252 (2006)
15. Hancock, P., Pattinson, D., Ghani, N.: Representations of stream processors using nested fixed points. Logical Methods in Comput. Sci. 5(3) (2009)
16. Mendler, N.P.: Inductive types and type constraints in the second-order lambda calculus. Ann. of Pure and Appl. Logic 51(1-2), 159–172 (1991)
17. Miculan, M.: On the formalization of the modal μ-Calculus in the Calculus of Inductive Constructions. Inform. and Comput. 164(1), 199–231 (2001)
18. Nakata, K., Uustalu, T.: Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: an exercise in mixed induction-coinduction. In: Aceto, L., Sobocinski, P. (eds.) Proc. of 7th Wksh. on Structural Operational Semantics, SOS 2010, Electron. Proc. in Theor. Comput. Sci., vol. 32, pp. 57–75 (2010)
19. Raffalli, C.: L' Arithmétiques Fonctionnelle du Second Ordre avec Points Fixes. PhD thesis, Université Paris VII (1994)
20. Sprenger, C.: A Verified Model Checker for the Modal μ-calculus in Coq. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998)
21. Tsai, M.-H., Wang, B.-Y.: Formalization of CTL * in Calculus of Inductive Constructions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 316–330. Springer, Heidelberg (2008)
22. Troelstra, A.S., van Dalen, D.: Constructivism in Mathematics, vol. I, II. North-Holland (1988)

A Semantics for Context-Sensitive Reduction Semantics

Casey Klein[1], Jay McCarthy[2], Steven Jaconette[1], and Robert Bruce Findler[1]

[1] Northwestern University
[2] Brigham Young University

Abstract. This paper explores the semantics of the meta-notation used in the style of operational semantics introduced by Felleisen and Hieb. Specifically, it defines a formal system that gives precise meanings to the notions of contexts, decomposition, and plugging (recomposition) left implicit in most expositions. This semantics is not naturally algorithmic, so the paper also provides an algorithm and proves a correspondence with the declarative definition.

The motivation for this investigation is PLT Redex, a domain-specific programming language designed to support Felleisen-Hieb-style semantics. This style of semantics is the de-facto standard in operational semantics and, as such, is widely used. Accordingly, our goal is that Redex programs should, as much as possible, look and behave like those semantics. Since Redex's first public release more than seven years ago, its precise interpretation of contexts has changed several times, as we repeatedly encountered reduction systems that did not behave according to their authors' intent. This paper describes the culimation of that experience. To the best of our knowledge, the semantics given here accommodates even the most complex uses of contexts available.

1 Introduction

The dominant style of operational semantics in use today has at its heart the notion of a context that controls where evaluation occurs. These contexts allow the designer of a reduction semantics to factor the definition of a calculus into one part that specifies the atomic steps of computation and a second part that controls where these steps may occur. This factoring enables concise specification, e.g., that a language is call-by-value or call-by-name or call-by-need (Ariola and Felleisen 1997), that `if` expressions must evaluate the test position before the branches, and even that exceptions, continuations, and state (Felleisen and Hieb 1992) behave in the expected ways, all without cluttering the rules that describe the atomic steps of computation.

Unfortunately, the precise meaning of context decomposition has not been nailed down in a way that captures its diverse usage in the literature. Although an intuitive definition is easy to understand from a few examples, this intuition does not cover the full power of contexts. For example, which terms match the pattern $C[e]$ from this language, in which values and contexts are mutually referential?

$$\begin{aligned} C &::= [] \mid (v\ C) \mid (C\ e) \\ e &::= v \mid \mathsf{x} \mid (e\ e) \\ v &::= (\lambda\ (\mathsf{x})\ e) \mid (\mathsf{cont}\ C) \end{aligned}$$

H. Yang (Ed.): APLAS 2011, LNCS 7078, pp. 369–383, 2011.

And which terms match this bizarre, small language?

$$C ::= C[(\mathsf{f}\ [])] \mid []$$

To remedy this lack, we have developed a semantics for matching and reduction that not only supports these exotic languages but also captures the intuitive meanings of countless existing research papers. This semantics does not assume explicit language-specific definitions of plugging and decomposition, since most expositions leave these concepts implicit.

Our motivation for studying context-sensitive matching is its implementation in the domain-specific programming language Redex (Felleisen et al. 2010; Matthews et al. 2004). Redex is designed to support the semantics engineer with a lightweight toolset for operational semantics and related formalisms. Specifically, Redex supports rapid prototyping of context-sensitive operational semantics, random testing, automatic typesetting, and, via its embedding in Racket, access to a large palette of standard programming tools. Redex is widely used, having supported several dozen research papers as well as the latest Scheme standard (Sperber et al. 2007) and a number of larger models, including one of the Racket virtual machine (Klein et al. 2010).

In keeping with the spirit of Redex, we augment a standard proof-based validation of our work with testing. More concretely, in addition to proving a correspondence between a specification of context-sensitive matching and an algorithm for that specification, we have conducted extensive testing of the semantics, using a Redex model of Redex (there is little danger of meta-circularity causing problems, as the embedding uses a modest subset of Redex's functionality—notably, no contexts or reduction relations). This model allows us to test that our semantics gives the intended meanings to interesting calculi from the literature, something that would be difficult to prove.

The remainder of this paper builds up an intuitive understanding of what contexts are and how they are used via a series of examples, gives a semantics for Redex's rewriting system, and discusses an algorithm to implement the semantics.

2 Matching and Contexts

This section introduces the notion of contexts and explains through a series of examples how matching works in their presence. Each example comes with a lesson that informs the design of our context-sensitive reduction semantics semantics.

In its essence, a pattern of the form $C[e]$ matches an expression when the expression can be split into two parts, an outer part (the context) that matches C and an inner part that matches e. The outer part marks where the inner part appears with a hole, written $[]$. In other words, when thinking of an expression as a tree, matching against $C[e]$ finds some subtree of the expression that matches e, and then replaces that sub-term with a hole to build a new expression in such a way that the new expression matches C.

$$a ::= (+\ a\ a) \mid number$$
$$C ::= (+\ C\ a) \mid (+\ a\ C) \mid []$$

$$C[(+\ number_1\ number_2)] \longrightarrow C[\Sigma[[number_1, number_2]]]$$

Fig. 1. Arithmetic Expressions

To get warmed up, consider figure 1. In this language a matches addition expressions and C matches contexts for addition expressions. More precisely, C matches an addition expression that has exactly one hole. For example, the expression (+ 1 2) matches $C[a]$ three ways, as shown in figure 2. Accordingly, the reduction relation given in figure 1 reduces addition expressions wherever they appear in an expression, e.g., reducing (+ (+ 1 2) (+ 3 4)) to two different expressions, (+ 3 (+ 3 4)) and (+ (+ 1 2) 7). This example tells us that our context matching semantics must support multiple decompositions for any given term.

A common use of contexts is to restrict the places where reduction may occur in order to model a realistic programming language's order of evaluation. Figure 3 gives a definition of E that enforces call-by-value left-to-right order of evaluation. For example, consider this nested set of function calls, $((f\ x)\ (g\ y))$, in which the result of $(g\ y)$ is passed to the result of $(f\ x)$. It decomposes into the context $([]\ (g\ y))$, allowing evaluation in the first position of the application. It does not, however, decompose into the context $((f\ x)\ [])$, since the grammar for E allows the hole to appear in the argument position of an application expression only when the function position is already a value. Accordingly, the reduction system insists that the call to f happens before the call to g. This example tells us that our semantics for decomposition must be able to support multiple different ways to decompose each expression form, depending on the subexpressions of that form (application expressions in this case).

$C = []$	$a = (+\ 1\ 2)$
$C = (+\ []\ 2)$	$a = 1$
$C = (+\ 1\ [])$	$a = 2$

Fig. 2. Example Decomposition

Contexts can also be used in clever ways to model the call-by-need λ-calculus. Like call-by-name, call-by-need evaluates the argument to a function only if the value is actually needed by the function's body. Unlike call-by-name, each function argument is evaluated at most once. A typical implementation of a language with call-by-need uses state to track if an argument has been evaluated, but it is also possible to give a direct explanation, exploiting contexts to control where evaluation occurs.

$$
\begin{aligned}
e &::= (e\ e) \mid x \mid v \\
v &::= (\lambda\ (x)\ e) \mid +1 \mid \mathit{number} \\
E &::= (E\ e) \mid (v\ E) \mid []
\end{aligned}
$$

Fig. 3. λ-calculus

Figure 4 shows the contexts from Ariola and Felleisen (1997)'s model of call-by-need. The first three of E's alternatives are standard, allowing evaluation in the argument of the +1 primitive, as well as in the function position of an application (regardless of what appears in the argument position). The fourth alternative allows evaluation in the body of a λ-expression that is in the function position of an application. Intuitively, this case says that once we have determined the function to be applied, then we can begin to evaluate its body. Of course, the function may eventually need its argument, and at that point, the final alternative comes into play. It says that when an applied function needs its argument, then that argument may be evaluated.

$$
\begin{aligned}
E ::= {} & [] \mid (+1\ E) \mid (E\ e) \\
& \mid ((\lambda\ (x)\ E)\ e) \\
& \mid ((\lambda\ (x)\ E[x])\ E)
\end{aligned}
$$

Fig. 4. Call-by-need Contexts

As an example, the expression ((λ (*x*) (+1 1)) (+1 2)) reduces by simplifying the body of the λ-expression to 2, without reducing the argument, because it decomposes into this context ((λ (*x*) []) (+1 2)) using the fourth alternative of E. In contrast, ((λ (*x*) (+1 *x*)) (+1 2)) reduces to ((λ (*x*) (+1 *x*)) 3) because the body of the λ-expression decomposes into the context (+1 []) with *x* in the hole, and thus the entire expression decomposes into the context ((λ (*x*) (+1 *x*)) []). This use of contexts tells us that our semantics must be able to support a sophisticated form of nesting, namely that sometimes a decomposition must occur in one part of a term in order for a decomposition to occur in another.

When building a model of first-class continuations, there is an easy connection to make, namely that an evaluation context is itself a natural representation for a continuation. That is, at the point that a continuation is grabbed, the context in which it is grabbed is the continuation. Figure 5 extends the left-to-right call-by-value model in figure 3 with support for continuations. It adds call/cc, the operator that grabs a continuation, and the new value form (cont E) that represents a continuation and can be applied to invoke the continuation.

$$v ::= .... \mid \text{call/cc} \mid (\text{cont}\ E)$$
$$E[(\text{call/cc}\ v)] \longrightarrow E[(v\ (\text{cont}\ E))]$$
$$E_1[((\text{cont}\ E_2)\ v)] \longrightarrow E_2[v]$$

Fig. 5. Continuations

For example, the expression (+1 (call/cc (λ (k) (k 2)))) reduces by grabbing a continuation. In this model that continuation is represented as (cont (+1 [])), which is then applied to call/cc's argument in the original context, yielding the expression (+1 ((λ (k) (k 2)) (cont (+1 [])))). The next step is to substitute for k, which yields the expression (+1 ((cont (+1 [])) 2)). This expression has a continuation value in the function position of an application, making the next step invoke the continuation. So, we can simply replace the context of the continuation invocation with the context inside the continuation, plugging the argument passed to the continuation in the hole, yielding (+1 2). This reduction system tells us that our context decomposition semantics must be able to support contexts that appear in a term that play no part in any decomposition (and yet must still match a specified pattern, such as E).

Generalizing from ordinary continuations to delimited continuations is simply a matter of factoring the contexts into two parts, one that may contain prompts and one that may not. Figure 6 shows one way to do this, as an extension of the call-by-value lambda calculus from figure 3.

$$e ::= .... \mid (\#\ e)$$
$$v ::= .... \mid \text{call/comp} \mid (\text{comp}\ M)$$
$$M ::= (M\ e) \mid (v\ M) \mid []$$
$$E ::= M \mid E[(\#\ M)]$$

$$E[(\#\ M[(\text{call/comp}\ v)])] \longrightarrow E[(\#\ M[(v\ (\text{comp}\ M))])]$$

Fig. 6. Delimited Continuations

The non-terminal E matches an arbitrary evaluation context and M matches an evaluation context that does not contain any prompt expressions. Accordingly, the rule for grabbing a continuation exploits this factoring to record only the portion of the context between the call to call/comp and the nearest enclosing prompt.

The interesting aspect of this system is how E refers to M and how that makes it difficult to support an algorithm that matches E. For all of the example systems in this section so far, a matching algorithm can match a pattern of the form $C[e]$ by attempting to match C against the entire term and, once a match has been found, attempting to match what appeared at the hole against e. With E, however, this leads to an infinite loop because E expands to a decomposition that includes E in the first position.[1]

A simple fix that works for the delimited continuations example is to backtrack when encountering such cycles; that fix, however, does not work for the first definition of C given in figure 7. Specifically, C would match only [] with an algorithm that treats that cycle as a failure to match, but the context $(f\ [])$ should match C, and more generally, the two definitions of C in figure 7 should be equivalent.

$$C ::= C[(\mathsf{f}\ [])] \mid []$$
$$C ::= (\mathsf{f}\ C) \mid []$$

Fig. 7. Wacky Context

3 A Semantics for Matching

This section formalizes the notion of matching used in the definitions of the example reduction systems in section 2. For ease of presentation, we stick to the core language of patterns and terms in figure 8. Redex supports a richer language of patterns (notably including a notion of Kleene star), but this core captures an essence suitable for explaining the semantics of matching.

Ignoring embedded contexts, a term t is simply a binary tree where leaf nodes are atoms a and interior nodes are constructed with cons. A context C is similarly a binary tree, but with a distinguished path (marked with left and right) from the root of the context to its hole.

Contexts are generated by decomposition and represent single-holed contexts. Although the hole can appear multiple times in a single term, such terms represent expressions that contain multiple, independently pluggable contexts.

$$\begin{array}{rcl} t & ::= & (\mathsf{cons}\ t\ t) \\ & \mid & a \\ & \mid & C \\ C & ::= & \mathsf{hole} \\ & \mid & (\mathsf{left}\ C\ t) \\ & \mid & (\mathsf{right}\ t\ C) \\ p & ::= & a \\ & \mid & (\mathsf{name}\ x\ p) \\ & \mid & (\mathsf{nt}\ n) \\ & \mid & (\mathsf{in\text{-}hole}\ p\ p) \\ & \mid & (\mathsf{cons}\ p\ p) \\ & \mid & \mathsf{hole} \\ a & \in & \mathit{Literals} \\ x & \in & \mathit{Variables} \\ n & \in & \mathit{Non\text{-}Terminals} \end{array}$$

Fig. 8. Patterns and Terms

Patterns p take one of six forms. Atomic patterns a and the hole pattern match only themselves. A pattern $(\mathsf{name}\ x\ p)$ binds the pattern variable x to the term matched by p. Repeated pattern variables force the corresponding sub-terms to be identical.

[1] Some find the equivalent, non-problematic grammar $E ::= M \mid M[(\#\ E)]$ clearer. At least one author of the present paper (who has spent a considerable amount of time hacking on Redex's implementation, no less), however, does not and was surprised when Redex failed to terminate on a similar example. We have also received comments from Redex users who were surprised by similar examples, suggesting that Redex should support such definitions.

A pattern (nt n) matches terms that match any of the alternatives of the non-terminal n (defined outside the pattern). We write decomposition patterns $p_1[p_2]$ using a separate keyword for clarity: (in-hole p_1 p_2). Finally, interior nodes are matched by the pattern (cons p_1 p_2), where p_1 and p_2 match the corresponding sub-terms.

For example, the left-hand side of the reduction rule in figure 1 corresponds to the following pattern, where the literal empty is used for the empty sequence and the pattern number matches literal numbers:

```
(in-hole (name C (nt C))
         (cons +
               (cons (name number_1 number)
                     (cons (name number_2 number)
                           empty))))
```

Figure 9 gives a semantics for patterns via the judgment form $G \vdash t : p \mid b$, which defines when the pattern p matches the term t. The grammar G is a finite map from non-terminals to sets of patterns. The bindings b is a finite map from pattern variables to terms showing how the pattern variables of p can be instantiated to yield t. The $G \vdash t : p \mid b$ judgment relies on an auxiliary judgment $G \vdash t = C[t'] : p \mid b$ that performs decompositions. Specifically, it holds when t can be decomposed into a context C that matches p and contains the sub-term t' at its hole.

Many of the rules for these two judgment forms rely on the operator $\sqcup$. It combines two mappings into a single one by taking the union of their bindings, as long as the domains do not overlap. If the domains do overlap, then the corresponding ranges must be the same; otherwise $\sqcup$ is not defined. Accordingly, rules that use $\sqcup$ apply only when $\sqcup$ is well-defined.

The $G \vdash t : p \mid b$ rules are organized by the structure of p. The atom and hole rules produce an empty binding map because those pattern contain no pattern variables. The name rule matches p with t and produces a map extended with the binding (x, t). The nt rule applies if any of the non-terminal's alternatives match. The scope of an alternative's pattern variables is limited to that alternative, and consequently, the nt rule produces an empty binding map. The cons rule matches the sub-terms and combines the resulting sets of bindings. The in-hole rule uses the decomposition judgment form to find a decomposition and checks that the term in the hole matches p_2.

The rules for the $G \vdash t = C[t'] : p \mid b$ form are also organized around the pattern. The hole decomposition rule decomposes any term t into the empty context and t itself. The first of two cons decomposition rules applies when a decomposition's focus may be placed within a pair's left sub-term. This decomposition highlights the same sub-term t_1' as the decomposition of t_1 does, but places it within the larger context (left C t_2). The second of the cons decomposition rules does the same for the pair's right sub-term. The nt decomposition rule propagates decompositions but, as in the corresponding matching rule, ignores binding maps.

The in-hole decomposition rule performs a nested decomposition. Nested decomposition occurs, for example, when decomposing according to call-by-need evaluation contexts (see the last production in figure 4). The in-hole rule decomposes t into a

$$\boxed{G \vdash t : p \mid b} \quad \frac{}{G \vdash a : a \mid \varnothing} \qquad \frac{}{G \vdash \mathsf{hole} : \mathsf{hole} \mid \varnothing}$$

$$\frac{G \vdash t : p \mid b}{G \vdash t : (\mathsf{name}\ x\ p) \mid \{(x, t)\} \sqcup b}$$

$$\frac{p \in G(n) \qquad G \vdash t : p \mid b}{G \vdash t : (\mathsf{nt}\ n) \mid \varnothing}$$

$$\frac{G \vdash t_1 : p_1 \mid b_1 \qquad G \vdash t_2 : p_2 \mid b_2 \qquad k \in \{\mathsf{cons}, \mathsf{left}, \mathsf{right}\}}{G \vdash (k\ t_1\ t_2) : (\mathsf{cons}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{G \vdash t_1 = C[t_2] : p_1 \mid b_1 \qquad G \vdash t_2 : p_2 \mid b_2}{G \vdash t_1 : (\mathsf{in\text{-}hole}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\boxed{G \vdash t = C[t'] : p \mid b}$$

$$\frac{}{G \vdash t = \mathsf{hole}[t] : \mathsf{hole} \mid \varnothing}$$

$$\frac{G \vdash t_1 = C[t_1'] : p_1 \mid b_1 \qquad G \vdash t_2 : p_2 \mid b_2 \qquad k \in \{\mathsf{cons}, \mathsf{left}, \mathsf{right}\}}{G \vdash (k\ t_1\ t_2) = (\mathsf{left}\ C\ t_2)[t_1'] : (\mathsf{cons}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{G \vdash t_1 : p_1 \mid b_1 \qquad G \vdash t_2 = C[t_2'] : p_2 \mid b_2 \qquad k \in \{\mathsf{cons}, \mathsf{left}, \mathsf{right}\}}{G \vdash (k\ t_1\ t_2) = (\mathsf{right}\ t_1\ C)[t_2'] : (\mathsf{cons}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{p \in G(n) \qquad G \vdash t_1 = C[t_2] : p \mid b}{G \vdash t_1 = C[t_2] : (\mathsf{nt}\ n) \mid \varnothing}$$

$$\frac{G \vdash t = C_1[t_1] : p_1 \mid b_1 \qquad G \vdash t_1 = C_2[t_2] : p_2 \mid b_2}{G \vdash t = (C_1 +\!\!+ C_2)[t_2] : (\mathsf{in\text{-}hole}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{G \vdash t_1 = C[t_2] : p \mid b}{G \vdash t_1 = C[t_2] : (\mathsf{name}\ x\ p) \mid \{(x, C)\} \sqcup b}$$

$G \in \mathit{Non\text{-}Terminal} \to \wp(p)$
$b \in \mathit{Variable} \to t$

$$\begin{aligned} \mathsf{hole} +\!\!+ C &= C \\ (\mathsf{left}\ C_1\ t) +\!\!+ C_2 &= (\mathsf{left}\ (C_1 +\!\!+ C_2)\ t) \\ (\mathsf{right}\ t\ C_1) +\!\!+ C_2 &= (\mathsf{right}\ t\ (C_1 +\!\!+ C_2)) \end{aligned}$$

Fig. 9. Matching and Decomposition

composed context C_1 ++ C_2 and a sub-term t', where p_1 and p_2 match C_1 and C_2 respectively. The definition of context composition (figure 9, bottom-right) follows the path in C_1. The name decomposition rule is similar to the corresponding matching rule, but it introduces a binding to the context that is matched, not the entire term.

4 An Algorithm for Matching

The rules in figure 9 provide a declarative definition of context-sensitive matching, but they do not lead directly to a tractable matching algorithm. There are two problems. First, as reflected in the two cons decomposition rules, an algorithm cannot know a priori whether to match on the left and decompose on the right or to decompose on the left and match on the right. An implementation that tries both possibilities scales exponentially in the number of nested cons patterns matched (counting indirect nesting through non-terminals). Second, the rules provide no answer to the question of whether to proceed in expanding a non-terminal if none of the input term has been consumed since last encountering that non-terminal. This question arises, for example, when decomposing by the non-terminal E from the grammar in figure 6, since E's second alternative causes the in-hole rule to decompose the same term by E. This second problem is the manifestation of left recursion in the form of grammars we consider.

The first problem can be solved by matching and decomposing simultaneously. Since these tasks differ only in their treatment of hole patterns, much work can be saved by sharing intermediate results between the tasks. Figure 10 demonstrates this approach with a function M that returns a set of pairs (d, b) representing possible ways to match or decompose the input term. In a pair representing a match, d is the marker •; in a pair representing a decomposition, d is a pair (C, t) such that the input term can be decomposed into a context C and a sub-term t occurring in C's hole.

The first two M cases handle the pattern hole. If the term in question is also hole, then it may be considered either to match hole or to decompose into hole in the empty context. If the term is not hole, then only decomposition is possible. The third case handles atomic patterns by producing a match result only if the given term is identical to the atom.

The (meta) context in which a call to M appears may eventually discard some or all of the results it receives. For example, consider the fourth clause, which handles cons patterns. If the term is also a pair (constructed with any of cons, left, or right), then this case makes two recursive calls and examines the cross product of the results using the select helper function. For each result pair, the case merges their bindings and checks that the results are not both decompositions. If neither is a decomposition, select combines the pair into a match result; if exactly one is a decomposition, it extends the decomposition with the term matched by the non-decomposition. If both are decompositions, then the match fails.

The next case, for patterns (in-hole p_c p_h), recurs with p_c and the input term, expecting to receive decompositions. For each one, it makes another recursive call, this time with p_h and the sub-term in the decomposition's focus. Each of the latter call's results m is combined with the decomposition's context, yielding a match result if m is a match and a larger context if m is a decomposition.

$$\begin{array}{ll}
\mathsf{matches} : G\ p\ t \to \wp(b) & m ::= (d, b) \\
\mathsf{matches}[\![G, p, t]\!] = \{b \mid (\bullet, b) \in \mathsf{M}[\![G, p, t]\!]\} & d ::= (C, t) \mid \bullet
\end{array}$$

$$\begin{array}{ll}
\mathsf{M} : G\ p\ t \to \wp(m) & \\
\mathsf{M}[\![G, \mathsf{hole}, \mathsf{hole}]\!] & = \{((\mathsf{hole}, \mathsf{hole}), \varnothing),\ (\bullet, \varnothing)\} \\
\mathsf{M}[\![G, \mathsf{hole}, t]\!] & = \{((\mathsf{hole}, t), \varnothing)\} \\
\mathsf{M}[\![G, a, a]\!] & = \{(\bullet, \varnothing)\} \\
\mathsf{M}[\![G, (\mathsf{cons}\ p_l\ p_r), (k\ t_l\ t_r)]\!] & = \{(d, b) \mid k \in \{\mathsf{cons},\ \mathsf{left},\ \mathsf{right}\}, \\
 & \qquad d \in \mathsf{select}[\![t_l, d_l, t_r, d_r]\!], \\
 & \qquad b = b_l \sqcup b_r, \\
 & \qquad (d_r, b_r) \in \mathsf{M}[\![G, p_r, t_r]\!], \\
 & \qquad (d_l, b_l) \in \mathsf{M}[\![G, p_l, t_l]\!]\} \\
\mathsf{M}[\![G, (\mathsf{in\text{-}hole}\ p_c\ p_h), t]\!] & = \{(d, b) \mid d = \mathsf{combine}[\![C, d_h]\!], \\
 & \qquad b = b_c \sqcup b_h, \\
 & \qquad (d_h, b_h) \in \mathsf{M}[\![G, p_h, t_c]\!], \\
 & \qquad ((C, t_c), b_c) \in \mathsf{M}[\![G, p_c, t]\!]\} \\
\mathsf{M}[\![G, (\mathsf{name}\ x\ p), t]\!] & = \{(d, b') \mid b' = \{(x, \mathsf{named}[\![d, t]\!])\} \sqcup b, \\
 & \qquad (d, b) \in \mathsf{M}[\![G, p, t]\!]\} \\
\mathsf{M}[\![G, (\mathsf{nt}\ n), t]\!] & = \{(d, \varnothing) \mid (d, b) \in \mathsf{M}[\![G, p, t]\!], p \in G(n)\} \\
\mathsf{M}[\![G, p, t]\!] & = \{\}
\end{array}$$

$$\begin{array}{ll}
\mathsf{select} : t\ d\ t\ d \to \wp(d) & \\
\mathsf{select}[\![t_1, \bullet, t_2, \bullet]\!] & = \{\bullet\} \\
\mathsf{select}[\![t, (C, t_1'), t_2, \bullet]\!] & = \{((\mathsf{left}\ C\ t_2), t_1')\} \\
\mathsf{select}[\![t_1, \bullet, t_2, (C, t_2')]\!] & = \{((\mathsf{right}\ t_1\ C), t_2')\} \\
\mathsf{select}[\![t_1, (C, t_1'), t_2, (C', t_2')]\!] & = \{\}
\end{array}$$

$$\begin{array}{ll}
\mathsf{combine} : C\ d \to d & \\
\mathsf{combine}[\![C, \bullet]\!] & = \bullet \\
\mathsf{combine}[\![C_1, (C_2, t)]\!] & = (C_1 +\!\!+ C_2, t)
\end{array}$$

$$\begin{array}{ll}
\mathsf{named} : d\ t \to t & \\
\mathsf{named}[\![\bullet, t]\!] & = t \\
\mathsf{named}[\![(C, t_1), t_2]\!] & = C
\end{array}$$

Fig. 10. Core matching algorithm (cases apply in order)

The remaining three cases are straightforward. The name case recurs on the sub-pattern and extends the bindings of each of the results with either the matched term or the context carved out by the decomposition. The nt case tries each alternative, discarding the binding component of each result. The final case, a catch-all, applies when the pattern does not match or decompose the input term.

Putting aside the problem of left recursion, the call M[[G, p, t]] computes the set of b such that $G \vdash t : p \mid b$ or $G \vdash t = C[t'] : p \mid b$ for some C and t', and the top-level wrapper function matches restricts this set to the bindings associated with match derivations.

To make this precise, we first give a definition of left-recursion. Intuitively, a grammar is left-recursive if there is a way, in a straight-forward recursive parser, to get from some non-terminal back to that same non-terminal without consuming any input. So, our definition of left-recursion builds a graph from the grammar by connecting each pattern to the other patterns that might reached without consuming any input, and then checks for a cycle in the graph. The most interesting case is the last one, where an in-hole pattern is connected to its second argument when the first argument can generate hole.

Definition. A grammar G is *left recursive* if $p \rightarrow_G^* p$ for some p, where $\rightarrow_G^*$ is the transitive (but not reflexive) closure of $\rightarrow_G \subseteq p \times p$, the least relation satisfying the following conditions:

1. (**nt** n) $\rightarrow_G p$ if $p \in G(n)$,
2. (**name** $n\ p$) $\rightarrow_G p$.
3. (**in-hole** $p\ p'$) $\rightarrow_G p$,
4. (**in-hole** $p\ p'$) $\rightarrow_G p'$ if $G \vdash$ **hole** : $p \mid b$, and

Theorem. For all G, p, and t, if G is not left recursive, then $b \in$ **matches**$[\![G, p, t]\!] \Leftrightarrow G \vdash t : p \mid b$.

The complete proof is available at `eecs.northwestern.edu/~robby/plug/`.

Parsing algorithms that support left recursive context-free grammars go back nearly fifty years (Kuno 1965). We refer the reader to Frost et al. (2007, section 3) for a summary. We have implemented an extension of the packrat parsing algorithm (Warth et al. 2008) that dynamically detects left recursion and treats the choice leading to it as a failure. If the other choices for the same portion of the input make any progress at all, the algorithm repeats the parse attempt, in hopes that the entries added to the memo table during the failed attempt will cause a second attempt to succeed. This process continues as long as repeated attempts make additional progress. Extending the algorithm in figure 10 with a similar iterative phase allows matching of terms from left recursive grammars, such the ones in figure 6 and figure 7.

5 A Semantics for Reduction

We now put the notion of matching from section 3 to work in a formalization of the standard notation for context-sensitive reduction rules. As with patterns, we consider

$$\frac{G \vdash t : p \mid b \qquad t' = \mathsf{inst}[[r, b]]}{G \vdash t \,/\, p \longrightarrow t' \,/\, r}$$

$$\begin{array}{l} r ::= a \\ \quad \mid \mathsf{hole} \\ \quad \mid (\mathsf{var}\ x) \\ \quad \mid (\mathsf{app}\ f\ r) \\ \quad \mid (\mathsf{in\text{-}hole}\ r\ r) \\ \quad \mid (\mathsf{cons}\ r\ r) \\ f \in t \to t \end{array}$$

$$\begin{array}{lll} \mathsf{inst} : r\ b \to t & & \\ \mathsf{inst}[[a, b]] & = & a \\ \mathsf{inst}[[\mathsf{hole}, b]] & = & \mathsf{hole} \\ \mathsf{inst}[[(\mathsf{var}\ x), b]] & = & b(x) \\ \mathsf{inst}[[(\mathsf{in\text{-}hole}\ r_1\ r_2), b]] & = & \mathsf{plug}[[\mathsf{inst}[[r_1, b]], \mathsf{inst}[[r_2, b]]]] \\ \mathsf{inst}[[(\mathsf{cons}\ r_1\ r_2), b]] & = & \mathsf{join}[[\mathsf{inst}[[r_1, b]], \mathsf{inst}[[r_2, b]]]] \\ \mathsf{inst}[[(\mathsf{app}\ f\ r), b]] & = & \delta(f, \mathsf{inst}[[r, b]]) \end{array}$$

$$\begin{array}{lll} \mathsf{plug} : C\ t \to t & & \\ \mathsf{plug}[[\mathsf{hole}, t]] & = & t \\ \mathsf{plug}[[(\mathsf{left}\ C_l\ t_r), C]] & = & (\mathsf{left}\ \mathsf{plug}[[C_l, C]]\ t_r) \\ \mathsf{plug}[[(\mathsf{left}\ C_l\ t_r), t]] & = & (\mathsf{cons}\ \mathsf{plug}[[C_l, t]]\ t_r) \\ \mathsf{plug}[[(\mathsf{right}\ t_l\ C_r), C]] & = & (\mathsf{right}\ t_l\ \mathsf{plug}[[C_r, C]]) \\ \mathsf{plug}[[(\mathsf{right}\ t_l\ C_r), t]] & = & (\mathsf{cons}\ t_l\ \mathsf{plug}[[C_r, t]]) \end{array}$$

$$\begin{array}{lll} \mathsf{join} : t\ t \to t & & \\ \mathsf{join}[[C, t]] & = & (\mathsf{left}\ C\ t) \quad \text{where } \mathsf{no\text{-}ctxts}\ t \\ \mathsf{join}[[t, C]] & = & (\mathsf{right}\ t\ C) \quad \text{where } \mathsf{no\text{-}ctxts}\ t \\ \mathsf{join}[[t_1, t_2]] & = & (\mathsf{cons}\ t_1\ t_2) \end{array}$$

$\delta : (t \to t)\ t \to t$
An unspecified function that applies metafunctions

$$\frac{}{\mathsf{no\text{-}ctxts}\ a} \qquad \frac{\mathsf{no\text{-}ctxts}\ t_1 \qquad \mathsf{no\text{-}ctxts}\ t_2}{\mathsf{no\text{-}ctxts}\ (\mathsf{cons}\ t_1\ t_2)}$$

Fig. 11. A semantics for reduction (function cases apply in order)

a core specification language that lacks many of the conveniences of a language like Redex but nevertheless highlights the principal ideas.

Figure 11 shows our definition. A user of Redex specifies a grammar and rules of the shape $p \longrightarrow r$, each consisting of a pattern p and a term template r. Redex uses the judgment form in the upper-left corner of the figure to determine if a particular term t reduces to t' by the given rule. The grammar in the figure's top-right gives the syntax for term templates, which include atoms, the context hole, references to variables bound by the left-hand side, applications of meta-level functions (e.g., substitution), hole-filling operations, and pairing operations.

The rest of the figure defines template instantiation. Atoms and hole instantiate to themselves, variables instantiate to their values, and meta-applications instantiate to the result of applying the meta-function to the instantiated argument template.

The instantiation of in-hole templates makes use of a generic plug function that accepts a context and a term and returns the result of plugging the context with the term. When plug's second argument is a context, it constructs a larger context by concatenating the two contexts, preserving the path to the hole. The path extension is necessary, for example, to support the following rule for an unusual control operator:

$$E[(\text{call/cc2 } v)] \longrightarrow E[(v\ (\text{cont } E[E]))]$$

When plug's second argument is some non-context term, it replaces the left or right constructor with cons, producing a non-context term.

Insistence that plug's first argument be a context creates a potential problem for rules which extend contexts, like this one for another unusual control operator:

$$E[(\text{call/cc+ } v)] \longrightarrow E[(v\ (\text{cont } (\text{+1 } E)))]$$

Although the rule does not explicitly define a path for the extended context $(\text{1+ } E)$, one can be safely inferred, since the term paired with E has no pluggable sub-terms.

The case of the inst function for cons templates performs this inference via the function join. When given a context and a term containing no contexts, join extends the context's path through the extra layer. When both arguments contain contexts, join combines the terms with cons, preventing possible ambiguity in a subsequent plugging operation.

Note, however, that the embedded contexts themselves remain pluggable by reduction rules and meta-functions that later pick apart the result term. For example, consider the rule for yet another unusual control operator:

$$E[(\text{call/ccs } v)] \longrightarrow E[(v\ (\text{tuple } (\text{cont } E)\ (\text{cont } E[E])))]$$

This rule calls v with a pair of continuation values. The term denoting this pair is not itself pluggable, but the embedded contexts can be plugged by subsequent reduction steps, after they are extracted by the reduction rules for projecting tuple components.

In addition to these contrived reduction rules, the semantics in figure 11 supports all of the systems in section 2, as well as the most sophisticated uses of contexts we have encountered in the literature, specifically:

- Ariola and Felleisen (1997)'s core call-by-need calculus. Their extension of this calculus to `letrec` uses decomposition in fundamentally the same way, but the particular formulation they choose makes use of pattern-matching constructs orthogonal to the ones we describe here, namely associative-commutative matching and a Kleene star-like construct that enforces dependencies between adjacent terms. The examples directory distributed with Redex shows one way to define their `letrec` evaluation contexts without these constructs, which Redex does not currently support.
- Flatt et al. (2007)'s semantics for delimited control in the presence of dynamic binding, exception handling, and Scheme's `dynamic-wind` form.
- Chang and Felleisen (2011)'s call-by-need calculus, which defines evaluation contexts using a heavily left-recursive grammar.

6 Related Work

Barendregt (1984) makes frequent use of a notion of contexts specialized to λ-terms. Like ours, these contexts may contain multiple holes, but plug's behavior differs in that it fills all of the context's holes. Felleisen and Hieb (1992) exploit the power of a selective notion of context to give equational semantics for many aspects of programming languages, notably continuations and state. The meaning of multi-holed grammars does not arise in their work, since the grammar for contexts restricts them to exactly one hole.

Lucas (1995) later explored an alternative formulation of selective contexts. This formulation defines contexts not by grammars but by specifying, for each function symbol, which sub-term positions may be reduced. Because the specification depends only on the function symbol's identity (i.e., and not on its sub-terms), this formulation cannot express common evaluation strategies, such as left-to-right, call-by-value evaluation. Follow-up work on this form of context-sensitive rewriting focuses on tools for proving termination, generally a topic of limited interest when studying reduction systems designed to model a programming language since these systems are not expected to terminate.

As part of their work on SL, a meta-language similar to Redex, Xiao et al. (2001) define a semantics for Felleisen-Hieb contexts by translating grammars into their own formalism, an extension of finite tree automata. This indirect approach allows SL to prove decomposition lemmas automatically using existing automata algorithms, but it is considerably more complicated than our approach and does not allow for multi-holed grammars like the ones in figure 5 and figure 6.

Dubois (2000) develops the first formulation of a Felleisen-Hieb reduction semantics in a proof assistant, as part of a mechanized proof of the soundness of ML's type system. Her formulation encodes single-hole contexts as meta-level term-to-term functions (restricted to coincide with the usual grammar defining call-by-value evaluation) and therefore models plug as meta-application. The formulation does not use an explicit notion of decomposition; instead, the contextual closure reduction rule applies to terms that may be formed using the plug operation.

Berghofer's, Leroy's, and Xi's solutions to the POPLmark Challenge (Aydemir et al. 2005) use Dubois's encoding for the challenge's reduction semantics. Vouillon's solution uses a first-order encoding of contexts and therefore provides an explicit definition

of plugging. The other submitted solutions use structural operational semantics, do not address dynamic semantics at all, or are no longer available online.

Danvy and Nielsen (2004) and Sieczkowski et al. (2010) provide an axiomatization of the various components of a Felleisen-Hieb reduction semantics, such as a decomposition relation, that together define the semantics. This axiomatization is not an appropriate basis for Redex for two reasons. First, it requires users to specify plugging and decomposition explicitly. Common practice leaves these definitions implicit, and one of our design goals for Redex is to support conventional definitions. Second, the axiomatization requires decomposition to be a (single-valued) function, ruling out the semantics in figure 1 and, more problematically, reduction semantics for multi-threaded programs and programs in languages like C and Scheme, which do not specify an order of evaluation for application expressions.

More broadly speaking, there are hundreds[2] of papers that use evaluation context semantics to model programming languages for just as many different purposes. Although we have not implemented anywhere near all of them in Redex, we have sought out interesting and non-standard ones over the years to try them out and to build our intuition about how a semantics should behave.

Acknowledgments. Thanks to Stephen Chang for his many interesting examples of contexts that challenged our understanding of context-sensitive matching. Thanks also to Matthias Felleisen and Matthew Flatt for helpful discussions of the work.

A version of this paper can be found online at:

`http://www.eecs.northwestern.edu/~robby/plug/`

That web page contains the final version of the paper as it appears in the proceedings and the Redex models for all of the figures in this paper.

References

Ariola, Z.M., Felleisen, M.: The Call-by-Need Lambda-Calculus. J. Functional Programming 7(3), 265–301 (1997)

Aydemir, B.E., Bohannon, A., Fairbairn, M., Nathan Foster, J., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized Metatheory for the Masses: The POPLmark Challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)

Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. North-Holland (1984)

Chang, S., Felleisen, M.: The Call-by-need Lambda Calculus (unpublished manuscript, 2011)

Danvy, O., Nielsen, L.R.: Refocusing in Reduction Semantics. Aarhus University, BRICS RS-04-26 (2004)

Dubois, C.: Proving ML Type Soundness within Coq. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 126–144. Springer, Heidelberg (2000)

[2] There are more than 400 citations to the original Felleisen-Hieb paper; while evaluation-context based semantics are still widely used, the paper is now rarely cited as it has become a standard part of the programming languages landscape.

Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2010)
Felleisen, M., Hieb, R.: The Revised Report on the Syntactic Theories of Squential Control and State. Theoretical Computer Science 103(2), 235–271 (1992)
Flatt, M., Yu, G., Findler, R.B., Felleisen, M.: Adding Delimited and Composable Control to a Production Programming Environment. In: Proc. ACM Intl. Conf. Functional Programming, pp. 165–176 (2007)
Frost, R.A., Hafiz, R., Callaghan, P.C.: Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars. In: Proc. International Conference on Parsing Technology, pp. 109–120 (2007)
Klein, C., Flatt, M., Findler, R.B.: The Racket Virtual Machine and Randomized Testing (2010), http://plt.eecs.northwestern.edu/racket-machine/
Kuno, S.: The Predictive Analyzer and a Path Elimination Technique. Communications of the ACM 8(7), 453–462 (1965)
Lucas, S.: Fundamentals of Context-Sensitive Rewriting. In: Bartosek, M., Staudek, J., Wiedermann, J. (eds.) SOFSEM 1995. LNCS, vol. 1012, pp. 405–412. Springer, Heidelberg (1995)
Matthews, J., Findler, R.B., Flatt, M., Felleisen, M.: A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 301–311. Springer, Heidelberg (2004)
Sieczkowski, F., Biernacka, M., Biernacki, D.: Automating Derivations of Abstract Machines from Reduction Semantics: A Generic Formalization of Refocusing in Coq. In: Proc. Symp. Implementation and Application of Functional Languages. To appear in LNCS (2010)
Sperber, M., Kent Dybvig, R., Flatt, M., van Straaten, A., Kelsey, R., Clinger, W., Rees, J., Findler, R.B., Matthews, J.: Revised [6] Report on the Algorithmic Language Scheme. Cambridge University Press (2007)
Warth, A., Douglass, J.R., Millstein, T.: Packrat Parsers Can Support Left Recursion. In: Proc. ACM SIGPLAN Wksp. Partial Evaluation and Program Manipulation, pp. 103–110 (2008)
Xiao, Y., Sabry, A., Ariola, Z.M.: From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition. Higher-Order and Symbolic Computation 14(4), 387–409 (2001)

Author Index

Albert, Elvira 238
Alglave, Jade 272
Arenas, Puri 238
Askarov, Aslan 220

Baillot, Patrick 337
Berghofer, Stefan 89
Bezem, Marc 353
Bjørner, Nikolaj 4
Bodin, Martin 19
Boissinot, Benoit 137
Bonchi, Filippo 289
Brandner, Florian 137

Caballero, Rafael 66
Cao, Zhen 121
Chi, Yun-Yan 74

Dang, Thao 34
Darte, Alain 137
Dong, Yuan 121
Dupont de Dinechin, Benoît 137

Findler, Robert Bruce 369
Fronc, Lukasz 322

Gadducci, Fabio 289
García-Ruiz, Yolanda 66
Gawlitza, Thomas Martin 34
Genaim, Samir 238
Gómez-Zamalloa, Miguel 238
Gupta, Ashutosh 188

Huang, Wei 255

Jaconette, Steven 369
Jhala, Ranjit 3

Kameyama, Yukiyoshi 105
Klein, Casey 369
Kokaji, Yuichiro 105
Kroening, Daniel 272

Lugton, John 272

Magazinius, Jonas 220
Malkis, Alexander 172
Mauborgne, Laurent 172
McCarthy, Jay 369
Milanova, Ana 255
Monniaux, David 19
Monreale, Giacoma Valentina 289
Morihata, Akimasa 204
Mu, Shin-Cheng 74

Nakata, Keiko 353
Nimal, Vincent 272
Nipkow, Tobias 89
Nori, Aditya V. 1

Oh, Hakjoo 50
O'Hearn, Peter W. 17

Pommereau, Franck 322
Popeea, Corneliu 188
Puebla, German 238

Rajamani, Sriram K. 1
Rastello, Fabrice 137
Rybalchenko, Andrey 188

Sabelfeld, Andrei 220
Sáenz-Pérez, Fernando 66
Schöpp, Ulrich 305
Sui, Yulei 155

Tautschnig, Michael 272
Traytel, Dmitriy 89

Uustalu, Tarmo 353

Wang, Shengyuan 121

Xue, Jingling 155

Ye, Sen 155
Yew, Pen-Chung 155
Yi, Kwangkeun 50