

Chris Hawblitzel  
Dale Miller (Eds.)

LNCS 7679

# Certified Programs and Proofs

Second International Conference, CPP 2012  
Kyoto, Japan, December 2012  
Proceedings



Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*TU Dortmund University, Germany*

Madhu Sudan

*Microsoft Research, Cambridge, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max Planck Institute for Informatics, Saarbruecken, Germany*

Chris Hawblitzel Dale Miller (Eds.)

# Certified Programs and Proofs

Second International Conference, CPP 2012  
Kyoto, Japan, December 13-15, 2012  
Proceedings



Springer

Volume Editors

Chris Hawblitzel

Microsoft Research Redmond, WA, USA

E-mail: [chris.hawblitzel@microsoft.com](mailto:chris.hawblitzel@microsoft.com)

Dale Miller

INRIA Saclay and LIX, Ecole Polytechnique, Palaiseau Cedex, France

E-mail: [dale@lix.polytechnique.fr](mailto:dale@lix.polytechnique.fr)

ISSN 0302-9743

e-ISSN 1611-3349

ISBN 978-3-642-35307-9

e-ISBN 978-3-642-35308-6

DOI 10.1007/978-3-642-35308-6

Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: Applied for

CR Subject Classification (1998): F.3.1, F.4.1, D.3.3, I.2.3, D.2.4, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

© Springer-Verlag Berlin Heidelberg 2012

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

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

*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](http://www.springer.com))

# Preface

This volume contains the proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP 2012) held during 13–15 December 2012 in Kyoto, Japan.

The CPP series of meetings aims to cover those topics in computer science and mathematics in which certification via formal techniques is crucial. This year's edition of CPP was co-located with APLAS 2012 (Asian Symposium on Programming Languages and Systems); similarly, CPP 2011 and APLAS 2011 were co-located in Taiwan. Both CPP 2011 and CPP 2012 took place in Asia in order to provide a focus point for the work on certification that is occurring there. The plan is to eventually locate CPP in Europe and North America as well as in Asia. A manifesto for CPP, written by Jean-Pierre Jouannaud and Zhong Shao, appears in the proceedings of CPP 2011 (LNCS 7086).

We are pleased that Gilles Barthe and Naoki Kobayashi accepted our invitation to be invited speakers for CPP 2012 and that Xavier Leroy and Greg Morrisett accepted to be keynote speakers addressing both APLAS 2012 and CPP 2012.

The program committee for CPP 2012 was composed of 18 researchers from 12 countries. We received a total of 37 submissions and eventually accepted 18 papers. Every submission was reviewed by at least 4 program committee members and their selected reviewers.

We wish to thank the program committee members and their reviewers for their efforts in helping to evaluate the submissions: it was a privilege to work with them. The EasyChair conference management system helped us to deal with all aspects of putting together our program. It was a pleasure to work with Jacques Garrigue, the General Chair for CPP 2012, and with Atsushi Igarashi and Ranjit Jhala who were, respectively, the General Chair and the Program Committee Chair for APLAS 2012. We also wish to thank the invited speakers, the authors of submitted papers, and the reviewers for their interest and strong support of this new conference series. Finally, we thank Nagoya University Graduate School of Mathematics for institutional sponsorship of this meeting.

October 2012

Chris Hawblitzel  
Dale Miller

# Organization

## Organizing Committee

Jacques Garrigue  
Atsushi Igarashi

Nagoya University, Japan  
Kyoto University, Japan

## CPP Steering Committee

Andrew Appel  
Nikolaj Bjørner  
Georges Gonthier  
John Harrison  
Jean-Pierre Jouannaud  
(Co-chair)  
Gerwin Klein  
Tobias Nipkow  
Zhong Shao (Co-chair)

Princeton University, USA  
Microsoft Research Redmond, USA  
Microsoft Research Cambridge, UK  
Intel Corporation, USA  
INRIA, France and Tsinghua University, China  
NICTA, Australia  
Technische Universität München, Germany  
Yale University, USA

## Program Committee

Stefan Berghofer  
Wei-Ngan Chin  
Adam Chlipala  
Mike Dodds  
Amy Felty  
Xinyu Feng  
Herman Geuvers

Secunet Security Networks AG, Germany  
National Univ. of Singapore, Singapore  
MIT, USA  
University of Cambridge, UK  
University of Ottawa, Canada  
Toyota Technological Institute at Chicago, USA  
Radboud University Nijmegen,  
The Netherlands

Robert Harper  
Chris Hawblitzel  
Gerwin Klein  
Laura Kovacs  
Rupak Majumdar  
Dale Miller  
Lawrence Paulson  
Frank Piessens  
Randy Pollack  
Bow-Yaw Wang  
Santiago Zanella Béguelin

Carnegie Mellon University, USA  
Microsoft Research Redmond, USA  
NICTA and UNSW, Australia  
TU Vienna, Austria  
UCLA, USA  
INRIA, France  
University of Cambridge, UK  
Katholieke Universiteit Leuven, Belgium  
University of Edinburgh, UK  
Academia Sinica, Taiwan  
IMDEA Software Institute, Spain

## Additional Reviewers

Avigad, Jeremy  
Bourke, Timothy  
Brotherston, James  
Bulwahn, Lukas  
Campbell, Brian  
Capretta, Venanzio  
Chang, Bor-Yuh Evan  
Charguéraud, Arthur  
Chaudhuri, Kaustuv  
Costea, Andreea  
David, Cristina  
Dixon, Lucas  
Ellison, Chucky  
Gherghina, Cristian  
Hoefner, Peter  
Hölzl, Johannes  
Jackson, Paul  
Kloos, Johannes  
Kozen, Dexter  
Krebbers, Robbert  
Le, Quang Loc  
Le, Ton-Chanh  
Mahboubi, Assia  
Matichuk, Daniel  
McKinna, James  
Memarian, Kayvan

Merz, Stephan  
Moskal, Michał  
Muehlberg, Jan Tobias  
Murray, Toby  
Nakata, Keiko  
Nigam, Vivek  
Norrish, Michael  
O'Connor, Russell  
Pichardie, David  
Platzer, André  
Schmidt, Renate  
Scott, Owens  
Sergey, Ilya  
Sewell, Thomas  
Smans, Jan  
Stampoulis, Antonis  
Starostin, Artem  
Ta, Quang-Trung  
Tahar, Sofiene  
Théry, Laurent  
Tiu, Alwen  
Tuerk, Thomas  
van der Weegen, Eelis  
Vogels, Frederic  
Wickerson, John  
Wiedijk, Freek

# Table of Contents

Scalable Formal Machine Models . . . . .	1
<i>Greg Morrisett</i>	
Mechanized Semantics for Compiler Verification . . . . .	4
<i>Xavier Leroy</i>	
Automation in Computer-Aided Cryptography: Proofs, Attacks and Designs . . . . .	7
<i>Gilles Barthe, Benjamin Grégoire, César Kunz, Yassine Lakhnech, and Santiago Zanella Béguelin</i>	
Program Certification by Higher-Order Model Checking . . . . .	9
<i>Naoki Kobayashi</i>	
A Formally-Verified Alias Analysis . . . . .	11
<i>Valentin Robert and Xavier Leroy</i>	
Mechanized Verification of Computing Dominators for Formalizing Compilers . . . . .	27
<i>Jianzhou Zhao and Steve Zdancewic</i>	
On the Correctness of an Optimising Assembler for the Intel MCS-51 Microprocessor . . . . .	43
<i>Dominic P. Mulligan and Claudio Sacerdoti Coen</i>	
An Executable Semantics for CompCert C . . . . .	60
<i>Brian Campbell</i>	
Producing Certified Functional Code from Inductive Specifications . . . . .	76
<i>Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois</i>	
The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing under One Roof . . . . .	92
<i>Lukas Bulwahn</i>	
Proving Concurrent Noninterference . . . . .	109
<i>Andrei Popescu, Johannes Hölzl, and Tobias Nipkow</i>	
Noninterference for Operating System Kernels . . . . .	126
<i>Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein</i>	
Compositional Verification of a Baby Virtual Memory Manager . . . . .	143
<i>Alexander Vaynberg and Zhong Shao</i>	



Shall We Juggle, Coinductively? .....	160
<i>Keisuke Nakano</i>	
Proof Pearl: Abella Formalization of $\lambda$ -Calculus Cube Property .....	173
<i>Beniamino Accattoli</i>	
A String of Pearls: Proofs of Fermat’s Little Theorem .....	188
<i>Hing-Lun Chan and Michael Norrish</i>	
Compact Proof Certificates for Linear Logic .....	208
<i>Kaustuv Chaudhuri</i>	
Constructive Completeness for Modal Logic with Transitive Closure ....	224
<i>Christian Doczkal and Gert Smolka</i>	
Rating Disambiguation Errors .....	240
<i>Andrea Asperti and Wilmer Ricciotti</i>	
A Formal Proof of Square Root and Division Elimination in Embedded Programs .....	256
<i>Pierre Neron</i>	
Coherent and Strongly Discrete Rings in Type Theory .....	273
<i>Thierry Coquand, Anders Mörberg, and Vincent Siles</i>	
Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives .....	289
<i>Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond</i>	
<b>Author Index</b> .....	305

# Scalable Formal Machine Models

Greg Morrisett

Harvard University  
greg@eecs.harvard.edu

**Abstract.** In the past few years, we have seen machine-checked proofs of relatively large software systems, including compilers and micro-kernels. But like all formal arguments, the assurance gained by these mechanical proofs is only as good as the models we construct of the underlying machine. I will argue that how we construct and validate these models is of vital importance for the research community. In particular, I propose that we develop domain-specific languages (DSLs) for describing the semantics of machines, and build interpretations of these DSLs in our respective proof-development systems. This will allow us to factor out and re-use machine semantics for everything from software to hardware.

## 1 Overview

Thirty years ago, the idea that we might build a real software component, such as a compiler, and construct a machine-checked proof that it was correct, was still a dream. There were some examples, such as the work of Milner and Weyrauch [1], yet those systems tended to be small examples for toy architectures.

In the intervening decades, the community has made great progress: Automated deduction techniques (e.g., SAT solvers and SMT provers) have improved tremendously. Proof development systems including ACL2, Coq, HOL, Isabelle, NuPRL, and PVS have become more powerful, more scalable, and much easier to use. At the same time, our need for strong assurance in safety and security-critical software systems has grown, driving down the cost of proofs relative to the benefits. These factors, amongst others, have led to shining examples of *usable*, real-world software components with detailed proofs of correctness, such as Leroy’s CompCert compiler [2] or Klein et al’s seL4 micro-kernel [3]. The size and complexity of these artifacts has led to a new kind of “proof engineering” where careful design is needed to ensure not only that the proofs can be done, but that they can be maintained as the software systems evolve.

Stepping back, what does it mean to prove a compiler or operating system correct? In each case, we must define a formal semantics for the software component (e.g., the dialect of C supported by CompCert or the API provided by seL4) as well as the underlying machine on which the software is intended to execute. Developing these semantic models takes substantial effort, akin to building an interpreter for a programming language or architecture. For the toy languages and machines of the past, this was not difficult, but for realistic languages (e.g., Java) and architectures (e.g., the x86), constructing a full semantic specification demands an almost overwhelming amount of detail.

These specifications are so big, that we are sure to get some details wrong, suggesting three things: First, we should avoid building lots of different, incomplete, and incompatible specifications (one for each research project) and instead work together to build robust models for important systems that transcend proof-development environments. Second, the design of our semantic models must support (efficient) execution so that we can test our models against real-world implementations. Finally, these models must be carefully constructed so that we can update them, whether due to a bug or natural evolution, and yet insulate, to the best of our ability, the proofs from these updates.

As an example of some work towards these goals, I will describe an on-going project that my colleague Gang Tan and I have been pursuing on formal models of instruction set architectures. Our primary goal is to build certified tools for enforcing security policies on x86 machine code, including Software Fault Isolation [4], Control-Flow Isolation [5], and variants of Typed Assembly Language [6]. To do so, we require a formal model of the syntax and semantics of x86 code, and while fragments of such models exist, none of them had enough detail that we could directly use them. Furthermore, some of the fragments were coded in ACL2, others in Isabelle, and others in Coq.

Consequently, we started to build a new model of the x86. Our design was broken into two major stages, one addressing decoding and the other addressing execution. Both stages were designed around domain-specific languages (DSLs) inspired by work on re-targeting compilers [7,8,9]. One DSL is used to describe the decoder and is similar to other parsing generators, but unlike say Yacc, comes equipped with a formal semantics that makes it easy to reason compositionally and declaratively about the relation between concrete and abstract syntax. Furthermore, it is carefully designed to support extraction of an efficient, table-driven parser. The execution stage of our x86 model is described via translation to a simple, RISC-like register transfer language. This RTL language is itself parameterized by a notion of machine state and comes equipped with an operational semantics, as well as other tools needed for both symbolic reasoning as well as validation.

Using these tools, we have constructed a semantic model for a significant fragment of the x86 (all of the integer and most of the floating-point instructions), and as described in a previous paper, used the model to prove the correctness of a tool for enforcing software fault isolation [10]. As expected, our model had (and probably still has) a number of bugs, but the ability to extract an executable model for testing has proved invaluable. Furthermore, we believe the modular design makes it relatively easy to add new features to the model, such as support for multiple-cores or alternative instruction sets. Finally, we have found the approach to describing behavior, based largely on translation to a small, orthogonal target language, works well for other kinds of machines, including the abstract machines used for high-level languages.

Nevertheless, in hindsight, there are many design decisions that we got wrong. For instance, we used a deep-embedding of our DSLs into Coq, taking advantage of features such as modules and dependent types that other proof assistants

may lack. Consequently, it is not clear how to “port” our specifications to other environments. As another example, we followed the design of compiler-based RTLs a little too closely and used an imperative representation for the intermediate code, when a functional representation makes both execution and symbolic reasoning easier.

## References

1. Milner, R., Weyhrauch, R.: Proving compiler correctness in a mechanized logic. In: *Proceedings of the 7th Annual Machine Intelligence Workshop*. Machine Intelligence, vol. 7, pp. 51–72. Edinburgh University Press (1972)
2. Leroy, X.: Formal verification of a realistic compiler. *Commun. of the ACM* 52(7), 107–115 (2009)
3. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: sel4: formal verification of an os kernel. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009*, pp. 207–220. ACM, New York (2009)
4. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: *Proc. of the 14th ACM Symp. on Operating Systems Principles, SOSP 1993*, pp. 203–216. ACM (1993)
5. Abadi, M., Budi, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: *Proc. of the 12th ACM Conf. on Computer and Commun. Security, CCS 2005*, pp. 340–353. ACM (2005)
6. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. In: *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 1998*, pp. 85–97. ACM (1998)
7. Ramsey, N., Fernandez, M.F.: Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.* 19(3), 492–524 (1997)
8. Ramsey, N., Davidson, J.W.: Machine Descriptions to Build Tools for Embedded Systems. In: Müller, F., Bestavros, A. (eds.) *LCTES 1998*. LNCS, vol. 1474, pp. 176–192. Springer, Heidelberg (1998)
9. Dias, J., Ramsey, N.: Automatically generating instruction selectors using declarative machine descriptions. In: *Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2010*, pp. 403–416. ACM (2010)
10. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: better, faster, stronger sfi for the x86. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012*, pp. 395–404. ACM, New York (2012)

# Mechanized Semantics for Compiler Verification

Xavier Leroy

INRIA Paris-Rocquencourt  
xavier.leroy@inria.fr

**Abstract.** The formal verification of compilers and related programming tools depends crucially on the availability of appropriate mechanized semantics for the source, intermediate and target languages. In this invited talk, I review various forms of operational semantics and their mechanization, based on my experience with the formal verification of the CompCert C compiler.

What does this program do, exactly? What is this program transformation or analysis supposed to do, exactly? Formal semantics is the art of providing mathematically-precise answers to these questions. It is a prerequisite to the verification of individual programs, and also to the specification (let alone verification) of programs that operate over other programs, such as static analyzers, program provers, code generators, and optimizing compilers.

Fundamental questions rarely have unique answers. Indeed, a great many different styles of semantics have been explored over the last 50 years, ranging from denotational to axiomatic to operational. In some application areas, *de facto* standards of semantics have emerged, such as labeled transition systems for concurrency, following Milner’s seminal work on CCS and the  $\pi$ -calculus [1,2], and small-step reduction semantics in the type systems community, following Wright and Felleisen’s preservation-and-progress pattern for type soundness proofs [3].

The landscape of programming languages research evolves quickly, renewing interest in other forms of semantics. For example, mechanization—formalizing semantics “on machine” with the help of interactive theorem provers, rather than “on paper”—is becoming standard practice in our field. The POPLmark challenge [4] showed that elementary semantic tools such as capture-avoiding substitution can be difficult to mechanize. On the other hand, the power of proof assistants makes it easier to work with semantic styles that are difficult to get right on paper, such as step-indexed logical relations [5] or definitional interpreters [6].

Another evolution worth noting is to formalize “real world” languages, such as C and Javascript, and to prove semantic properties that go beyond type safety, such as semantic preservation for a code generation or optimization algorithm. The reduction-based semantics that work so well to prove type safety for small languages such as IMP, Mini-ML or Featherweight Java can “burst at the seams” when applied to big, messy languages such as C. Likewise, relating the executions of two programs, before and after a code transformation, is fundamentally more

difficult than showing the preservation of a typing invariant throughout the execution of a single program.

In this talk, I survey some of these issues based on my personal experience with the formal verification of the CompCert C compiler [7]. As part of this effort, S. Blazy and I had to give mechanized semantics to 14 languages: a very large subset of ANSI C as the source language, assembly for the ARM, PowerPC and x86 machine architectures as target languages, and 10 intermediate languages that bridge the semantic gap between the source and target languages. This semantic engineering is a large part of the CompCert effort, first because these semantics appear prominently in the statement of compiler correctness that we prove, second because we had to change these semantics in essential ways throughout the development of CompCert, in order to prove stronger correctness statements and to accommodate progressively bigger subsets of ANSI C.

The first verifications were conducted against natural (big-step) semantics for the source language and most of the intermediate languages; only the target assembly language was in pure transition (small-step) style [8,9]. Natural semantics lived up to its name, resulting in relatively straightforward specifications for our languages, and helping us discover the main insights of the semantic preservation proofs. However, we quickly hit limitations of natural semantics, such as its inability to describe nonterminating executions.

The second iteration of CompCert, therefore, uses a combination of small-step transition semantics with explicit call stack for most of the intermediate languages [10], and of *coinductive big-step semantics* for the source language and the first intermediate languages. Coinductive big-step semantics, as introduced by Grall and Leroy [11], enable divergence to be described by coinductive inference rules that follow the structure of executions, like natural semantics does for termination.

We then wanted to account for unstructured control (the `goto` statement) and nondeterministic evaluation order of C, and also to make provisions for a future extension towards shared-memory concurrency—many features where big-step semantics is not appropriate. We therefore switched to small-step, labeled transition semantics for the source and intermediate languages with structured control. We found reduction semantics in the style of MiniML or Featherweight Java inadequate for compiler proofs, but succeeded in using *continuation-based semantics* as introduced by Appel and Blazy [12]. These semantics carefully separate the current sub-command under execution from the execution context in which it appears, with the context being represented “inside-out” as a continuation term. This style of operational semantics is reminiscent not only of the CEK abstract machine [13], but also of polarization and focusing in proof theory and in  $\lambda$ -calculus [14,15].

CompCert’s journey through the landscape of operational semantics has been rather tortuous, but led to the discovery of original forms of operational semantics along the way. Are we at the end of the path? It depends on the language features we would like to model in the future. For instance, giving semantics to program fragments (compilation units) and reasoning about separate compilation and linking

probably requires more compositional reasoning principles based on logical relations, in the style of Benton and Hur [16]. In all likelihood, the large-scale formal verification of compilers and static analyzer, as well as other emerging applications of semantics, will keep challenging the state of the art in semantics and exposing the need for new approaches and mechanizations.

## References

1. Milner, R.: *Communication and Concurrency*. Prentice-Hall (1990)
2. Milner, R.: *Communicating and Mobile Systems: the pi-Calculus*. Cambridge University Press (1999)
3. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. *Information and Computation* 115(1), 38–94 (1994)
4. Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: 35th symposium Principles of Programming Languages, pp. 3–15. ACM Press (2008)
5. Appel, A.W., McAllester, D.A.: An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems* 23(5), 657–683 (2001)
6. Danielsson, N.A.: Operational semantics using the partiality monad. In: International Conference on Functional Programming 2012, pp. 127–138. ACM Press (2012)
7. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* 52(7), 107–115 (2009)
8. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd Symposium Principles of Programming Languages, pp. 42–54. ACM Press (2006)
9. Blazy, S., Dargaye, Z., Leroy, X.: Formal Verification of a C Compiler Front-End. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)
10. Leroy, X.: A formally verified compiler back-end. *Journal of Automated Reasoning* 43(4), 363–446 (2009)
11. Leroy, X., Grall, H.: Coinductive big-step operational semantics. *Information and Computation* 207(2), 284–304 (2009)
12. Appel, A.W., Blazy, S.: Separation Logic for Small-Step *CMINOR*. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 5–21. Springer, Heidelberg (2007)
13. Felleisen, M., Friedman, D.P.: Control operators, the SECD machine and the  $\lambda$ -calculus. In: *Formal Description of Programming Concepts III*, pp. 131–141. North-Holland (1986)
14. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science* 410(46), 4747–4768 (2009)
15. Curien, P.-L., Munch-Maccagnoni, G.: The Duality of Computation under Focus. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 165–181. Springer, Heidelberg (2010)
16. Benton, N., Hur, C.K.: Biorthogonality, step-indexing and compiler correctness. In: International Conference on Functional Programming 2009, pp. 97–108. ACM Press (2009)

# Automation in Computer-Aided Cryptography: Proofs, Attacks and Designs

Gilles Barthe<sup>1</sup>, Benjamin Grégoire<sup>3</sup>, César Kunz<sup>1,2</sup>,  
Yassine Lakhnech<sup>4</sup>, and Santiago Zanella Béguelin<sup>5</sup>

<sup>1</sup> IMDEA Software Institute

<sup>2</sup> Universidad Politécnica de Madrid

<sup>3</sup> INRIA Sophia Antipolis-Méditerranée

<sup>4</sup> Université de Grenoble, France

<sup>5</sup> Microsoft Research

CertiCrypt [3] and EasyCrypt [2] are machine-checked frameworks for proving the security of cryptographic constructions. Both frameworks adhere to the game-based approach [9,6,8] to provable security [7], but revisit its realization from a formal verification perspective. More specifically, CertiCrypt and EasyCrypt use a probabilistic programming language pWHILE for expressing cryptographic constructions, security properties, and computational assumptions, and a probabilistic relational Hoare logic pRHL for justifying reasonings in cryptographic proofs. While both tools coincide in their foundations, they differ in their underlying technologies: CertiCrypt is implemented as a set of libraries in the Coq proof assistant, whereas EasyCrypt uses a verification condition generator for pRHL in combination with off-the-shelf SMT solvers and automated theorem provers. Over the last six years, we have used both tools to verify prominent examples of public-key encryption schemes, modes of operation, signature schemes, hash function designs, zero-knowledge proofs. Recently, we have also used both tools to certify the output of a zero-knowledge compiler [1].

The next challenge is to extend EasyCrypt with automated mechanisms for discovering proofs or attacks. As a first step in this direction, we have developed a front-end that searches for security proofs or attacks for public-key encryption schemes built from one-way trapdoor permutations and random oracles. Given a candidate scheme, the front-end first searches for attacks using a deducibility relation inspired from symbolic cryptography: if an attack is found, it outputs an attacker. If not, the front-end searches for game-based proofs that the scheme is secure: if a proof is found, it outputs a concrete security bound and an EasyCrypt script that can be verified independently. We have evaluated the applicability of the front-end on more than hundred variants of OAEP [5], a widely used padding scheme commonly used for strengthening RSA encryption: pleasingly, it proves most secure variants of OAEP and computes security bounds that match known bounds in many cases. In addition, we have used the front-end in combination with synthesis algorithms to explore the design space of the class of encryption schemes it covers. This has led to the discovery of ZAEP [4], a simplified variant of the OAEP padding scheme that can be used to strengthen RSA encryption with exponents 2 and 3.



More information about the project can be found at:

<http://easycrypt.gforge.inria.fr>

## References

1. Almeida, J.B., Barbosa, M., Bangerter, E., Barthe, G., Krenn, S., Béguelin, S.Z.: Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols. In: 19th ACM Conference on Computer and Communications Security, CCS 2012. ACM (2012)
2. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-Aided Security Proofs for the Working Cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
3. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 90–101. ACM, New York (2009)
4. Barthe, G., Pointcheval, D., Béguelin, S.Z.: Verified security of redundancy-free encryption from Rabin and RSA. In: 19th ACM Conference on Computer and Communications Security, CCS 2012. ACM (to appear, 2012)
5. Bellare, M., Rogaway, P.: Optimal Asymmetric Encryption. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 92–111. Springer, Heidelberg (1995)
6. Bellare, M., Rogaway, P.: The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006)
7. Goldwasser, S., Micali, S.: Probabilistic encryption. *J. Comput. Syst. Sci.* 28(2), 270–299 (1984)
8. Halevi, S.: A plausible approach to computer-aided cryptographic proofs. *Cryptology ePrint Archive*, Report 2005/181 (2005)
9. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. *Cryptology ePrint Archive*, Report 2004/332 (2004)

# Program Certification by Higher-Order Model Checking

Naoki Kobayashi

The University of Tokyo

**Abstract.** Model checking of higher-order recursion schemes or (collapsible) higher-order pushdown automata (higher-order model checking, for short) is a generalization of finite state and pushdown model checking, which has been extensively studied in the last decade [1–11, 15–17]. Higher-order recursion schemes are essentially terms of the simply-typed  $\lambda$ -calculus with recursion and tree constructors; therefore, they serve as good models for higher-order functional programs. Indeed, various verification problems for higher-order functional programs can be easily reduced to higher-order model checking, and automated verification tools have been developed based on the reduction [9, 12–14, 18].

In the talk, I will first provide a brief introduction to higher-order model checking and its applications to higher-order program verification. I will then discuss higher-order model checking from the viewpoint of certificates. In particular, I plan to discuss the following questions: (i) How can we certify the result of program verification based on higher-order model checking? (ii) Why does higher-order model checking work at all, despite its extremely high worst-case complexity?

## References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. *Logical Methods in Computer Science* 3(3) (2007)
2. Broadbent, C.H., Carayol, A., Hague, M., Serre, O.: A Saturation Method for Collapsible Pushdown Systems. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) *ICALP 2012, Part II*. LNCS, vol. 7392, pp. 165–176. Springer, Heidelberg (2012)
3. Broadbent, C.H., Carayol, A., Ong, C.-H.L., Serre, O.: Recursion schemes and logical reflection. In: *Proceedings of LICS 2010*, pp. 120–129. IEEE Computer Society Press (2010)
4. Carayol, A., Serre, O.: Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In: *Proceedings of LICS 2012*. IEEE Computer Society Press (2012)
5. Hague, M., Murawski, A., Ong, C.-H.L., Serre, O.: Collapsible pushdown automata and recursion schemes. In: *Proceedings of 23rd Annual IEEE Symposium on Logic in Computer Science*, pp. 452–461. IEEE Computer Society (2008)
6. Hague, M., Ong, C.-H.L.: Symbolic backwards-reachability analysis for higher-order pushdown systems. *Logical Methods in Computer Science* 4(4) (2008)
7. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-Order Pushdown Trees Are Easy. In: Nielsen, M., Engberg, U. (eds.) *FOSSACS 2002*. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)

8. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009, pp. 25–36. ACM Press (2009)
9. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL), pp. 416–428 (2009)
10. Kobayashi, N.: A Practical Linear Time Algorithm for Trivial Automata Model Checking of Higher-Order Recursion Schemes. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 260–274. Springer, Heidelberg (2011)
11. Kobayashi, N., Ong, C.-H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009, pp. 179–188. IEEE Computer Society Press (2009)
12. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 222–233 (2011)
13. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL), pp. 495–508 (2010)
14. Lester, M.M., Neatherway, R.P., Ong, C.-H.L., Ramsay, S.J.: Model checking liveness properties of higher-order functional programs. In: Proceedings of ML Workshop 2011 (2011)
15. Neatherway, R.P., Ramsay, S.J., Ong, C.-H.L.: A traversal-based algorithm for higher-order model checking. In: ACM SIGPLAN International Conference on Functional Programming (ICFP 2012), pp. 353–364 (2012)
16. Ong, C.-H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90. IEEE Computer Society Press (2006)
17. Ong, C.-H.L.: Models of higher-order computation: Recursive schemes and collapsible pushdown automata. In: Logics and Languages for Reliability and Security, pp. 263–299. IOS Press (2010)
18. Ong, C.-H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL), pp. 587–598 (2011)

# A Formally-Verified Alias Analysis

Valentin Robert<sup>1,2</sup> and Xavier Leroy<sup>1</sup>

<sup>1</sup> INRIA Paris-Rocquencourt

<sup>2</sup> University of California, San Diego

vrobot@cs.ucsd.edu, xavier.leroy@inria.fr

**Abstract.** This paper reports on the formalization and proof of soundness, using the Coq proof assistant, of an alias analysis: a static analysis that approximates the flow of pointer values. The alias analysis considered is of the points-to kind and is intraprocedural, flow-sensitive, field-sensitive, and untyped. Its soundness proof follows the general style of abstract interpretation. The analysis is designed to fit in the CompCert C verified compiler, supporting future aggressive optimizations over memory accesses.

## 1 Introduction


**Alias Analysis.** Most imperative programming languages feature *pointers*, or *object references*, as first-class values. With pointers and object references comes the possibility of *aliasing*: two syntactically-distinct program variables, or two semantically-distinct object fields can contain identical pointers referencing the same shared piece of data.

The possibility of aliasing increases the expressiveness of the language, enabling programmers to implement mutable data structures with sharing; however, it also complicates tremendously formal reasoning about programs, as well as optimizing compilation. In this paper, we focus on optimizing compilation in the presence of pointers and aliasing. Consider, for example, the following C program fragment:

```
... *p = 1; *q = 2; x = *p + 3; ...
```

Performance would be increased if the compiler propagates the constant 1 stored in `p` to its use in `*p + 3`, obtaining

```
... *p = 1; *q = 2; x = 4; ...
```

This optimization, however, is unsound if `p` and `q` can alias. Therefore, the compiler is allowed to perform this optimization only if a prior static analysis, called *alias analysis* or *pointer analysis*, establishes that the pointers `p` and `q` differ in all executions of the program. 

---

<sup>1</sup> More precisely, the static analysis needed here is called *may-alias analysis* and aims at proving that two pointers are always different. There also exists *must-alias analyses*, which aim at proving that two pointers are always identical, but we will not consider these analyses in this paper.

For another example, consider:

```
... *p = x; y = *q; ...
```

To give more time to the processor cache to perform the load from `q`, and therefore improve instruction-level parallelism, an instruction scheduling pass would elect to permute the load from `q` and the store to `p`, obtaining:

```
... y = *q; *p = x; ...
```

Again, this optimization is sound only if the compiler can establish that `p` and `q` never alias. Many other optimizations rely on the availability of nonaliasing information. It is fair to say that a precise and efficient alias analysis is an important component of any optimizing compiler.

**Compiler Verification.** Our aim, in this paper, is to equip the CompCert C compiler with a may-alias analysis, in order to enable this compiler to perform more aggressive optimizations over memory accesses. CompCert C is a moderately-optimizing C compiler, producing code for the ARM, PowerPC and x86 architectures [11]. The distinguishing feature of CompCert C is that it is *formally verified* using the Coq proof assistant: a formal, operational semantics is given to every source, intermediate and target language used in CompCert, and a proof of semantic preservation is conducted for every compilation pass. Composing these proofs, we obtain that the assembly code generated by CompCert executes as prescribed by the semantics of the source C program, therefore ruling out the possibility of miscompilation.

When an optimization pass exploits the results of a prior static analysis, proving semantic preservation for this pass requires us to first prove *soundness* of the static analysis: the results of the analysis must, in a sense to be made precise, be a safe over-approximation of the possible run-time behaviors of the program. This paper, therefore, reports on the proof of soundness of an alias analysis for the RTL intermediate language of the CompCert compiler. In keeping with the rest of CompCert, we use the Coq proof assistant both to program the alias analysis and to mechanize its proof of correctness.<sup>2</sup> This work is, to the best of our knowledge, the first mechanized verification of an alias analysis.

**The Landscape of Alias Analyses.** Like most published may-alias analyses (see Hind [8] for a survey), ours is of the *points-to* kind: it infers sets of facts of the form “this abstract memory location may contain a pointer to that other abstract memory location”. Existing alias analyses differ not only on their notions of abstract memory locations, but also along the following orthogonal axes:

- *Intraprocedural vs. interprocedural*: an intraprocedural analysis processes each function of the program separately, making no nonaliasing assumptions about the values of parameters and global variables at function entry.

---

<sup>2</sup> The Coq development is available at <http://gallium.inria.fr/~varobert/alias/>

An interprocedural analysis processes groups of functions, or even whole programs, and can therefore infer more precise facts at the entry point of a function when all of its call sites are known.

- *Flow-sensitivity*: a flow-sensitive analysis such as Andersen’s [1] takes the control flow of the program into account, and is able to infer different sets of facts for different program points. A flow-insensitive analysis such as Steensgaard’s [16] maintains a single set of points-to facts that apply to all program points. Consider for example:

... L1: p = &x;      ... L2: p = &y;      ...

A flow-sensitive analysis can tell that just after L1,  $p$  points only to  $x$ , and just after L2,  $p$  points only to  $y$ . A flow-insensitive analysis would conclude that  $p$  points to either  $x$  or  $y$  after both L1 and L2.

- *Field-sensitivity*: a field-sensitive analysis is able to maintain different points-to information for different fields of a compound data structure, such as a C `struct`. A field-insensitive analysis makes no such distinction between fields.
- *Type-based vs. untyped*: many alias analyses operate at the source-language level (e.g. C or Java) and exploit the static typing information of this language (e.g. `struct` declarations in C and `class` declarations in Java). Other analyses ignore static type information, either because it is unreliable (as in C with casts between pointer types or nondiscriminated unions) or because it is not available (analysis at the level of intermediate or machine languages).

These characteristics govern the precision/computational cost trade-off of the analysis, with intraprocedural being cheaper but less precise than interprocedural, flow-insensitive cheaper and less precise than flow-sensitive, and type-based cheaper and more precise than untyped.

The alias analysis that we proved correct in Coq is of the *points-to*, *intraprocedural*, *flow-sensitive*, *field-sensitive*, and *untyped* kind: untyped, because the RTL language it works on is untyped; flow-sensitive, because it instantiates a general framework for dataflow analyses that is naturally flow-sensitive; field-sensitive, for additional precision at moderate extra analysis costs; and intraprocedural, because we wanted to keep the analysis and its proof relatively simple. Our analysis is roughly similar to the one outlined by Appel [2, section 17.5] and can be viewed as a simplified variant of Andersen’s seminal analysis [1].

**Related Work.** The literature on may-alias analysis is huge; we refer the reader to Hind [8] for a survey, and only discuss the mechanized verification of these analyses. Many alias analyses are instances of the general framework of abstract interpretation. Bertot [3], Besson *et al.* [4], and Nipkow [15] develop generic mechanizations of abstract interpretation in Coq and Isabelle/HOL, but do not consider alias analysis among their applications. Dabrowski and Pichardie [7] mechanize the soundness proof of a data race analysis for Java bytecode, which includes a points-to analysis, flow-sensitive for local variables but flow-insensitive for heap contents. The analysis is formally specified but its implementation is not verified. Their soundness proof follows a different pattern than ours, relying

on an instrumented, alias-aware semantics that is inserted between the concrete semantics of Java bytecode and the static analysis.

**Outline.** The remainder of this paper is organized as follows. Section 2 briefly introduces the RTL intermediate language over which the alias analysis is conducted. Section 3 explains how we abstract memory locations and execution states. Section 4, then, presents the alias analysis as a forward dataflow analysis. Section 5 outlines its soundness proof. Section 6 discusses a data structure, finite maps with overlapping keys and weak updates, that plays a crucial role in the analysis. Section 7 reports on an experimental evaluation of our analysis. Section 8 concludes and discusses possible improvements.

## 2 The RTL Intermediate Language

Our alias analysis is conducted over the RTL intermediate language [12, section 6.1]. RTL stands for “register transfer language”. It is the simplest of the CompCert intermediate languages, and also the language over which optimizations that benefit from nonaliasing information are conducted. RTL represents functions as a control-flow graph (CFG) of abstract instructions, corresponding roughly to machine instructions but operating over pseudo-registers (also called “temporaries”). Every function has an unlimited supply of pseudo-registers, and their values are preserved across function call. In the following,  $r$  ranges over pseudo-registers and  $l$  over labels of CFG nodes.

Instructions:	$i ::= \text{nop}(l)$ $\quad   \text{op}(op, \vec{r}, r, l)$ $\quad   \text{load}(\kappa, mode, \vec{r}, r, l)$ $\quad   \text{store}(\kappa, mode, \vec{r}, r, l)$ $\quad   \text{call}(sig, (r \mid id), \vec{r}, r, l)$ $\quad   \text{tailcall}(sig, (r \mid id), \vec{r})$ $\quad   \text{cond}(cond, \vec{r}, l_{true}, l_{false})$ $\quad   \text{return} \mid \text{return}(r)$	no operation (go to $l$ ) arithmetic operation memory load memory store function call function tail call conditional branch function return
Control-flow graphs:	$g ::= l \mapsto i$	finite map
Functions:	$F ::= \{ \text{sig} = sig;$ $\quad \text{params} = \vec{r};$ $\quad \text{stacksize} = n;$ $\quad \text{entrypoint} = l;$ $\quad \text{code} = g \}$	parameters size of stack data block label of first instruction control-flow graph

Each instruction takes its arguments in a list of pseudo-registers  $\vec{r}$  and stores its result, if any, in a pseudo-register  $r$ . Additionally, it carries the labels of its possible successors. Each function has a stack data block, automatically allocated on function entry and freed at function exit, in which RTL producers can allocate local arrays, structs, and variables whose addresses are taken.

The dynamic semantics of RTL is given in small-step style as a transition relation between execution states. States are tuples  $\text{State}(\Sigma, g, \sigma, l, R, M)$  of a

call stack  $\Sigma$ , a CFG  $g$  for the function currently executing, a pointer  $\sigma$  pointing to its stack data block, a label  $l$  for the CFG node to be executed, a register state  $R$  and a memory state  $M$ . (See Leroy [12], section 6.1] for more details on the semantics.)

Register states  $R$  map pseudo-registers to their current values: the disjoint union of 32-bit integers, 64-bit floats, pointers, and a special `undef` value. Pointer values  $\mathbf{Vptr}(b, i)$  are composed of a block identifier  $b$  and an integer byte offset  $i$  within this block.

Memory states  $M$  map (block, offset, memory type) triples to values. (See [14,13] for a complete description of the CompCert memory model.) Distinct memory blocks are associated to 1- every global variable of the program, 2- the stack blocks of every function currently executing, and 3- the results of dynamic memory allocation (the `malloc` function in C), which is presented as a special form of the `call` RTL instruction.

### 3 Abstracting Memory Locations and Memory States

The first task of a points-to analysis is to partition the unbounded number of memory blocks that can appear during execution into a finite, tractable set of abstract blocks. Since our analysis is intraprocedural, we focus our view of the memory blocks on the currently-executing function, and distinguish the following classes of abstract blocks:

Abstract blocks:  $\hat{b} ::= \mathbf{Stack}$   
 $\quad \quad \quad | \mathbf{Globals}(\mathbf{Just} \ id) \mid \mathbf{Globals}(\mathbf{All})$   
 $\quad \quad \quad | \mathbf{Allocs}(\mathbf{Just} \ l) \mid \mathbf{Allocs}(\mathbf{All})$   
 $\quad \quad \quad | \mathbf{Other} \mid \top$

$\mathbf{Stack}$  denotes the stack block for the currently-executing function;  $\mathbf{Globals}(\mathbf{Just} \ id)$ , the block associated to the global variable  $id$ ;  $\mathbf{Allocs}(\mathbf{Just} \ l)$ , the blocks dynamically allocated (by `malloc`) at point  $l$  in the current function; and  $\mathbf{Other}$  all other blocks, including stack blocks and dynamically-allocated blocks of other functions.

The  $\mathbf{Stack}$  and  $\mathbf{Globals}(\mathbf{Just} \ id)$  classes correspond to exactly one concrete memory block each. Other classes can match several concrete blocks. For example, if a call to `malloc` at point  $l$  occurs within a loop, several concrete blocks are allocated, all matching  $\mathbf{Allocs}(\mathbf{Just} \ l)$ .

To facilitate static analysis, we also introduce summary abstract blocks:  $\mathbf{Globals}(\mathbf{All})$ , standing for all the global blocks;  $\mathbf{Allocs}(\mathbf{All})$ , standing for all the dynamically-allocated blocks of the current function; and  $\top$ , standing for all blocks. The inclusions between abstract blocks are depicted in Fig. 11.

Two abstract blocks that are not related by inclusion denote disjoint sets of concrete blocks. We write  $\hat{b}_1 \cap \hat{b}_2 = \emptyset$  in this case. If, for instance, the analysis tells us that the pseudo-registers  $x$  may point to  $\mathbf{Stack}$  and  $y$  to  $\mathbf{Allocs}(\mathbf{Just} \ 3)$ , we know that  $x$  and  $y$  cannot alias.

To achieve field sensitivity, our analysis abstracts pointer values not just as abstract blocks, but as abstract locations: pairs  $\hat{\ell} = (\hat{b}, \hat{i})$  of an abstract block  $\hat{b}$



and an abstract offset  $\hat{i}$ , which is either an integer  $i$  or  $\top$ , denoting a statically-unknown offset. We extend the notion of disjointness to abstract locations in the obvious way:

$$(\hat{b}_1, \hat{i}_1) \cap (\hat{b}_2, \hat{i}_2) = \emptyset \stackrel{\text{def}}{=} \hat{b}_1 \cap \hat{b}_2 = \emptyset \vee (\hat{i}_1 \neq \top \wedge \hat{i}_2 \neq \top \wedge \hat{i}_1 \neq \hat{i}_2)$$

For example, the analysis can tell us that  $x$  maps to the abstract location  $(\text{Stack}, 4)$  and  $y$  to the abstract location  $(\top, 0)$ . In this case, we know that  $x$  and  $y$  never alias, since these two abstract locations are disjoint even though the two abstract blocks  $\text{Stack}$  and  $\top$  are not.

For additional precision, our analysis manipulates *points-to sets*  $\hat{P}$ , which are finite sets  $\{\hat{\ell}_1, \dots, \hat{\ell}_n\}$  of abstract locations. For example, the empty points-to set denotes any set of values that can be integers or floats but not pointers; the points-to set  $\{(\top, \top)\}$  denotes all possible values; and the points-to set  $\{(\text{Globals}(\text{All}), \top), (\text{Other}, \top)\}$  captures the possible values for a function parameter, before the stack block and the dynamic blocks of the function are allocated.

Our points-to analysis, therefore, associates a pair  $(\hat{R}, \hat{M})$  to every program point  $l$  of every function, where  $\hat{R}$  abstracts the register states  $R$  at this point by a finite map from pseudo-registers to points-to sets, and  $\hat{M}$  abstracts the memory states  $M$  at this point by a map from abstract pointers to points-to sets.

## 4 The Alias Analysis

The alias analysis we consider is an instance of forward dataflow analysis. Given the points-to information  $(\hat{R}, \hat{M})$  just “before” program point  $l$ , a transfer function conservatively estimates the points-to information  $(\hat{R}', \hat{M}')$  “after” executing the instruction at  $l$ , and propagates it to the successors of  $l$  in the control-flow graph. Kildall’s worklist algorithm [9], then, computes a fixed point over all nodes of the control-flow graph. The transfer function is defined by a complex case analysis on the instruction at point  $l$ . We now describe a few representative cases.

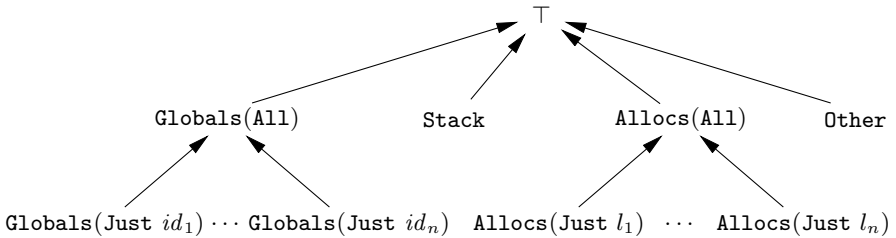


Fig. 1. Abstract blocks and their inclusion relation

For an arithmetic operation  $\text{op}(op, \vec{r}, r, l')$ , memory and pseudo-registers other than  $r$  are unchanged, therefore the points-to information “after” is  $(\hat{R}\{r \leftarrow \hat{P}\}, \hat{M})$ .  $\hat{P}$  is the abstraction of the result of the operation. Most operations compute integers or floats but not pointers, so we take  $\hat{P} = \emptyset$ . Other operations form pointers into the stack block or into global variables, where we take  $\hat{P} = \{\{\text{Stack}, i\}\}$  or  $\hat{P} = \{\{\text{Globals}(\text{Just } id), i\}\}$  as appropriate. Finally, for `move` instructions as well as pointer addition and pointer subtraction,  $\hat{P}$  is determined from the points-to sets  $\hat{R}(\vec{r})$  of the argument registers.

For a load instruction  $\text{load}(\kappa, mode, \vec{r}, r, l)$ , the points-to information “after” is, likewise, of the form  $(\hat{R}\{r \leftarrow \hat{P}\}, \hat{M})$ , where  $\hat{P}$  abstracts the value just loaded. If  $\kappa$  denotes a 32-bit integer-or-pointer quantity,  $\hat{P}$  is determined by querying the current abstract memory state  $\hat{M}$  at the abstract locations determined by the addressing mode  $mode$  applied to the points-to sets  $\hat{R}(\vec{r})$  of the argument registers. If  $\kappa$  denotes a small integer or floating-point quantity, the RTL semantics guarantee that the result of the load is not a pointer; we therefore take  $\hat{P} = \emptyset$ .

For a store instruction  $\text{store}(\kappa, mode, \vec{r}, r, l)$ , pseudo-registers are unchanged but memory is modified. The analysis determines the set  $L$  of abstract locations accessed, as a function of  $mode$  and  $\hat{R}(\vec{r})$ , then produces the points-to information  $(\hat{R}, \hat{M}')$ , where  $\hat{M}' = \hat{M} \sqcup \{\hat{\ell} \mapsto \hat{R}(r) \mid \hat{\ell} \in L\}$ .

Since our abstract pointers correspond, in general, to multiple concrete memory locations, we must perform a *weak update*: the points-to sets associated with  $\hat{\ell} \in L$  are not replaced by the points-to set  $\hat{R}(r)$ , but joined with the latter, using set union. Moreover, we must perform this weak update not only for the abstract locations in  $L$ , but also for all the abstract locations that are not disjoint from the locations in  $L$ . This weak update is achieved by our definition of the upper bound operation  $\sqcup$  over memory maps. The end result is that the new memory map  $\hat{M}'$  satisfies the following two properties that are crucial to our soundness proof:

$$\begin{aligned} \hat{M}'(\hat{\ell}) &\sqsupseteq \hat{M}(\hat{\ell}) && \text{for all abstract locations } \hat{\ell} \\ \hat{M}'(\hat{\ell}) &\sqsupseteq \hat{R}(r) && \text{if } \exists \hat{\ell}' \in L, \hat{\ell} \cap \hat{\ell}' \neq \emptyset \end{aligned}$$

As mentioned above, the alias analysis reuses the generic fixed-point solver provided by CompCert [12, section 7.1]. Termination is guaranteed by bounding the total number of iterations and returning  $\top$  if no fixpoint is reached within this limit. There is, therefore, no need to prove termination. However, the iteration limit is very high, therefore we must make sure that a fixpoint is reached relatively quickly. For such dataflow analyses, termination is typically ensured by the combination of two facts: the monotonicity of the transfer function and the finite height of the underlying lattice. While our transfer function is monotonic, our lattice of points-to sets does not have a finite height because the sets can grow indefinitely by adding more and more different abstract offsets. To address this issue, we ensure termination by *widening* [6], which accelerates possibly infinite chains by approximation: if the points-to set computed for some memory location or register, at an edge of the control flow graph, contains an accurate memory location (that is, an abstract block with a particular offset) which is

a shifted version (that is, the same abstract block and a different offset) of an element of the previous points-to set of that same object, then we widen that points-to set to contain the whole abstract block. This prevents any infinite chain of differing offsets. The lattice quotiented by this widening indeed has a finite height, since the number of abstract blocks to be considered within a function is bounded (the number of global variables and allocation sites is bounded).

## 5 Soundness Proof

The main contribution of this work is the mechanized proof that the alias analysis is *sound*: namely, that the properties of non-aliasing and flow of pointer values inferred by the analysis are satisfied in every possible execution of the analyzed program. The proof follows the general pattern of abstract interpretation, namely, establishing a correspondence between abstract “things” (blocks, locations, states, etc) and sets of concrete “things”, then show that this correspondence is preserved by transitions of the concrete semantics.

The correspondence is presented as relations between abstract and concrete “things”, parameterized by an *abstracting function*, called “abstracter” in the Coq development:

**Definition** `abstracter` := `block -> option absb`.

An abstracter maps every concrete memory block to an abstract memory block, or to `None` if the concrete block is not allocated yet. The abstracter is existentially quantified: the gist of the soundness proof is to construct a suitable abstracter for every reachable concrete execution state.

Given an abstracter, a concrete value belongs to a points-to set if the following predicate holds:

```

Definition valsat (v: val) (abs: abstracter) (s: PTSet.t) :=
  match v with
  | Vptr b o =>
    match abs b with
    | Some ab => PTSet.In (Loc ab o) s
    | None     => PTSet.ge s PTSet.top
    end
  | _           => True
  end.

```

In other words, non-pointer values belong to any points-to set. A pointer value `Vptr b o` belongs to the set `s` if the concrete block `b` is mapped to the abstract block `ab` by the abstracter and if the abstract location `Loc ab o`, or a “bigger” abstract location, appears in `s`. To simplify the proof, we also account for the case where `b` is not mapped by the abstracter, in which case we require `s` to contain all possible abstract locations, i.e. to be at least as large as the  $\top$  points-to set.

We extend the `valsat` relation to pseudo-registers and to memory locations. In the following, `(Rhat, Mhat)` are the register map and memory map computed by the static analysis at a given program point.

```
Definition regsat (r: reg) (rs: regset) (abs: abstracter) (Rhat: RMap.t) :=
  valsat rs#r abs (RMap.get r Rhat).
```

(The concrete value `rs#r` of register `r` belongs to the points-to set `RMap.get r Rhat` predicted by the analysis.)

```
Definition memsat (b: block) (o: Int.int) (m: mem)
  (abs: abstracter) (Mhat: MMap.t) :=
  forall v,
  Mem.loadv Mint32 m (Vptr b o) = Some v ->
  match abs b with
  | Some ab => valsat v abs (MMap.get (Loc ab o) Mhat)
  | None    => False
end).
```

(If, in the concrete memory state, location `(b, o)` contains value `v`, it must be the case that `b` is abstracted to `ab` and `v` belongs to the points-to set `MMap.get (Loc ab o) Mhat` predicted by the analysis.)

Not all abstracters are sound: they must map the stack block for the currently-executing function to the abstract block `Stack`, and the blocks for global variables to the corresponding abstract blocks:

```
Definition ok_abs_genv (abs: abstracter) (ge: genv) :=
  forall id b,
  Genv.find_symbol ge id = Some b ->
  abs b = Some (Just (Globals (Just id))).
```

It must also be the case that only valid, already-allocated concrete blocks are abstracted:

```
Definition ok_abs_mem (abs: abstracter) (m: mem) :=
  forall b, abs b <> None <-> Mem.valid_block m b.
```

Piecing everything together, we obtain the following characterization of concrete execution states that agree with the predictions of the static analysis:

```
Inductive satisfy (ge: genv) (abs: abstracter): state -> Prop :=
| satisfy_state: forall cs f bsp pc rs m Rhat Mhat
  (STK: ok_stack ge (Mem.nextblock m) cs)
  (MEM: ok_abs_mem abs m)
  (GENV: ok_abs_genv abs ge)
  (SP: abs bsp = Some (Just Stack))
  (RPC: (safe_funanalysis f)#pc = (Rhat, Mhat))
  (RSAT: forall r, regsat r rs abs Rhat)
  (MSAT: forall b o, memsat b o m abs Mhat),
  satisfy ge abs (State cs f (Vptr bsp Int.zero) pc rs m)
```

We omit the `ok_stack` predicate, which collects some technical conditions over the call stack `cs`. The `safe_funanalysis` function is the implementation of our alias analysis: it returns a map from program points `pc` to abstract states `(Rhat, Mhat)`.

In essence, the `satisfy` property says that the abstracter `abs` is sound (premises `MEM`, `GENV`, `SP`) and that, with respect to this abstracter, the values of registers and memory locations belong to the points-to sets predicted by the analysis at the current program point `pc` (premises `RPC`, `RSAT` and `MSAT`).

The main proof of soundness, then, is to show that for every concrete state `st` reachable during the execution of the program, the property `exists abs, satisfy ge abs st` holds:

```
Theorem satisfy_init:
  forall p st,
    initial_state p st ->
      exists abs, satisfy (Genv.globalenv p) abs st.
Theorem satisfy_step:
  forall ge st t st' abs,
    satisfy ge abs st -> step ge st t st' ->
      exists abs', satisfy ge abs' st'.
```

As a corollary, we obtain the soundness of the non-aliasing predictions made on the basis of the results of the analysis:

```
Corollary nonaliasing_sound:
  forall ge abs cs f sp pc rs m Rhat Mhat r1 b1 o1 r2 b2 o2,
    satisfy ge abs (State cs f sp pc rs m) ->
      (safe_funanalysis f)#pc = (Rhat, Mhat) ->
      disjoint (RMap.get r1 Rhat) (RMap.get r2 Rhat) ->
      rs # r1 = Vptr b1 o1 -> rs # r2 = Vptr b2 o2 ->
      Vptr b1 o1 <> Vptr b2 o2.
```

Here, `disjoint` is the decidable predicate stating that two sets of abstract locations are pairwise disjoint, in the sense of the  $\hat{\ell}_1 \cap \hat{\ell}_2 = \emptyset$  definition above. An optimization that exploits the inferred aliasing information would test whether `disjoint` holds of the points-to sets of two registers `r1` and `r2`. If the test is positive, and since the `satisfy` predicate holds at any reachable state, the corollary above shows that `r1` and `r2` do not alias at run-time, i.e. they cannot contain the same pointer value. In turn, this fact can be used in the proof of semantic preservation for the optimization.

The Coq development consists of about 1200 lines of specifications and 2200 lines of proofs. The proof is entirely constructive: given a suitable abstracter `abs` “before” a transition of the semantics, it is always possible to construct the abstracter `abs'` that satisfies the state after the transition. A large part of the proof is devoted to proving the many required properties of points-to sets and memory maps. A crucial invariant to be maintained is that memory maps  $\hat{M}$  maps stay compatible with the inclusion relation between abstract locations:  $\hat{M}(\hat{\ell}_1) \subseteq \hat{M}(\hat{\ell}_2)$  whenever  $\hat{\ell}_1 \sqsubseteq \hat{\ell}_2$ . For instance, the points-to set of the abstract block that represents all possible concrete blocks must be a superset of the points-to set of any abstract pointer. We maintain this invariant through the use of dependent types (Coq’s subset types).

Additional complications stem from the need to keep the representation of abstract memory states relatively small, eliminating redundant information in order to speed up map updates. We describe our solution in the next section.

## 6 Maps with Weak Update

Purely-functional finite maps are among the most frequently used data structures in specifications and programs written using proof assistants. The standard signature for total finite maps is of the following form:

```
Module Type Map (K: DecidableType) (V: AnyType).
  Parameter t: Type.
  Parameter init: V.t -> t
  Parameter get: K.t -> t -> V.t
  Parameter set: K.t -> V.t -> t -> t
  Axiom get_init: forall k v, get k (init v) = v
  Axiom get_set_same:
    forall k1 k2 v m, K.eq k1 k2 -> get k1 (set k2 v m) = v
  Axiom get_set_other:
    forall k1 k2 v m, ~K.eq k1 k2 -> get k1 (set k2 v m) = get k1 m
End Map.
```

Here,  $K$  is the type of map keys, equipped with a decidable equality, and  $V$  is the type of map values. Three operations are provided: `init`, to create a constant map; `set`, to change the value associated with a given key; and `get`, to obtain the value associated with a key. The semantics of `set` are specified by the familiar “good variable” properties `get_set_same` and `get_set_other` above. Such a signature of total finite maps can easily be implemented on top of an implementation of partial finite maps, such as the AVL maps provided by the Coq library `FMaps`.

To implement the memory maps inferred by our alias analysis, we need a slightly different finite map structure, where the strong update operation `set` is replaced by a weak update operation `add`. During weak update, not only the value of the updated key changes, but also the values of the keys that overlap with / are not disjoint from the updated key. Moreover, the new values of the changed keys are an upper bound of their old value and the value given to `add`. This is visible in the following signature:

```
Module Type OverlapMap (O: Overlap) (L: SEMILATTICE).
  Parameter t: Type.
  Parameter init: t.
  Parameter get: O.t -> t -> L.t.
  Parameter add: O.t -> L.t -> t -> t.
  Axiom get_init: forall x, get x init = L.bot.
  Axiom get_add:
    forall x y s m, L.ge (get x (add y s m)) (get x m).
  Axiom get_add_overlap: forall x y s m,
    O.overlap x y -> L.ge (get x (add y s m)) s.
End OverlapMap.
```

The type of map values, `L.t` is now a *semi-lattice*: a type equipped with a partial ordering `ge`, an upper bound operation `lub`, and a smallest element `bot`. Likewise, the type of keys, `O.t` is a type equipped with a decidable `overlap` relation, which holds when two keys are not disjoint. (Additional operations such as `parent` are included to support the efficient implementation that we discuss next.) Here is the `Overlap` signature:

```
Module Type Overlap.
  Parameter t: Type.
  Parameter eq_dec: forall (x y: t), {eq x y} + {~eq x y}.
  Parameter overlap: t -> t -> Prop.
  Axiom overlap_dec: forall x y, {overlap x y} + {~ overlap x y}.
  Declare Instance overlap_refl: Reflexive overlap.
  Declare Instance overlap_sym: Symmetric overlap.
  Parameter top: t.
  Parameter parent: t -> option t.
  Parameter measure: t -> nat.
  Axiom parent_measure: forall x px,
    parent x = Some px -> measure px < measure x.
  Axiom no_parent_is_top: forall x, parent x = None <-> x = top.
  Axiom parent_overlap: forall x y px,
    overlap x y -> parent x = Some px -> overlap px y.
End Overlap.
```

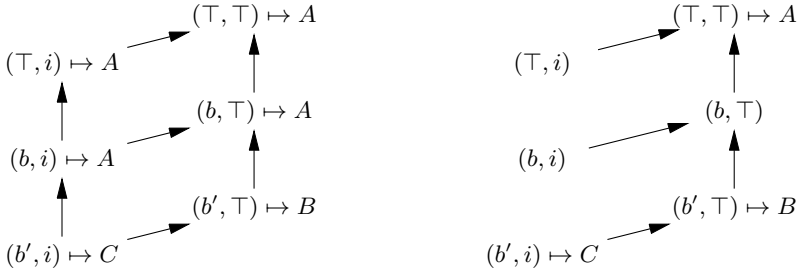
Note that the `overlap` relation must be reflexive and symmetric, but is not transitive in general. For example, in our application to alias analysis, `(Stack, T)` and `(Other, T)` do not overlap, but both overlap with `(T, T)`.

How, then, to implement the `OverlapMap` data structure? Naively, we could build on top of a regular `Map` data structure, implementing the `add` (weak update) operation by a sequence of `set` (strong updates) over all keys  $k_1, \dots, k_n$  that overlap the given key  $k$ . However, the set of overlapping keys is not necessarily finite: if  $k$  is `Allocs(All)`, all keys `Allocs(Just l)` overlap. Even if we could restrict ourselves to the program points  $l$  that actually occur in the function being analyzed, the set of overlapping keys would still be very large, resulting in inefficient `add` operations.

In practice, during alias analysis, almost all the keys `Allocs(Just l)` have the same values as the summary key `Allocs(All)`, except for a small number of distinguished program points  $l$ , and likewise for keys `Globals(Just id)`. This observation suggests a sparse representation of maps with overlap where we do not store the value of a key if this value is equal to that of its *parent* key.

More precisely, we assume that the client of the `OverlapMap` structure provides us with a *spanning tree* that covers all possible keys. This tree is presented as the `top` element of the `Overlap` structure, representing the root of the tree, and the `parent` partial function, which maps non-`top` keys to their immediate ancestor in the spanning tree. Fig. 2 depicts this spanning tree and the sparse representation in the case where abstract locations are used as keys.

Following these intuitions, we implement the type `t` of `OverlapMap` as a standard, partial, finite map with strong update (written `M` below), as provided by



**Fig. 2.** Sparse maps indexed by abstract locations. Left: the logical view. Arrows represent the inclusion relation between abstract locations. Each location is associated with a value. Right: the concrete representation. Arrows represent the parent relation (spanning tree). Some locations are not associated with a value, because their value is to be looked up in their parent locations.

Coq’s `FMap` library for example. We then define the `get` operation as a recursive traversal of the spanning tree, starting at the given key and moving “up” in the tree until a binding is found:

```
Function get (k: 0.t) (m: t) {measure 0.measure k}: L.t :=
  match M.find k m with
  | Some s => s
  | None => match 0.parent k with None => L.bot | Some p => get p m end
end.
```

The `add weak` update operation, then, can be defined by traversing all the non-default bindings found in the sparse map, and updating those that overlap with the given key:

```
Definition lub_if_overlap (key: 0.t) (val: L.t) (k: 0.t) (v: L.t): L.t :=
  if 0.overlap_dec key k then L.lub val v else v.
```

```
Definition add (k: 0.t) (v: L.t) (m: t): t :=
  M.mapi (lub_if_overlap k v) (M.add k (get k m) m).
```

Here, `M.mapi` is pointwise application of a function to a finite map: the map returned by `M.mapi f m` maps  $k$  to  $f k v$  if and only if  $m$  maps  $k$  to  $v$ .

The initial call to `M.add` is redundant if the key is already present, but necessary when the key is absent, in order to populate the underlying map with the key, at its current value, before performing the traversal. This definition almost satisfies the two “weak good variables” properties `get_add` and `get_add_overlap`: for the latter property, we need to assume that the `0.top` key is bound in the sparse map, otherwise some keys could keep their default `L.bot` value after the weak update. This assumption is easily satisfied by defining the initial map `init` not as the empty sparse map, but as the singleton sparse map `0.top ↦ L.bot`. To make sure that the assumption always holds, we package it along with the sparse maps using a subset type of the form

```
Definition t := { m : M.t | M.In 0.top m }
```



This makes it possible to prove the two “weak good variables” properties without additional hypotheses.

This sparse representation of maps, while simple, appears effective for our alias analysis. Two improvements can be considered. One would be to compress the sparse map after each `add` operation, removing the bindings  $k \mapsto v$  that have become redundant because  $k$ 's parent is now mapped to the same value  $v$ . Another improvement would be to enrich the data structure to ensure that non-overlapping keys have their values preserved by an `add` update:

```
Conjecture get_add_non_overlap: forall x y s m,
  ~0.overlap x y -> get x (add y s m) = get x m.
```

This property does not hold for our simple sparse representation: assume  $x$  not bound in the sparse map, its parent  $px$  bound in the sparse map,  $x$  non overlapping with  $y$ , but  $px$  overlapping with  $y$ . The value of  $px$  is correctly updated by the `add y s m` operation, but as a side effect this also modifies the result of `get x` after the `add`.

## 7 Experimental Evaluation

The first author integrated the alias analysis described here in the CompCert verified compiler and modified its Common Subexpression Elimination pass to exploit the inferred nonaliasing information. CompCert’s CSE proceeds by value numbering over extended basic blocks [12, section 7.3]. Without aliasing information, value numbering equations involving memory loads are discarded when reaching a memory write. Using aliasing information, CSE is now able to preserve such equations across memory writes whenever the address of the read is known to be disjoint from that of the write.

This implementation was evaluated on the CompCert test suite. Evaluation consisted in 1- visual examination of the points-to sets inferred to estimate the precision of the analysis, 2- measurements on compilation times, and 3- counting the number of instructions eliminated by CSE as a consequence of the more precise analysis of memory loads enabled by nonaliasing information.

Concerning precision, the analysis succeeds in inferring the expected nonaliasing properties between global and local variables of array or structure types, and between their fields. The lack of interprocedural analysis results in a very conservative analysis of linked, dynamically-allocated data structures, however.

Concerning analysis times, the cost of the analysis is globally high: on most of our benchmarks, overall compilation times increase by 40% when alias analysis is turned on, but a few files in the SPASS test exhibit pathological behaviors of the analysis, more than doubling compilation times.

The additional nonaliasing information enables CSE to eliminate about 1400 redundant loads in addition to the 5600 it removes without this information, a 25% gain. To illustrate the effect, here is an excerpt of RTL intermediate code before (left column) and after (right column) aliasing-aware CSE:

```

16: x16 = int8signed[currentCodeLen + 0]
15: x15 = x16 + -8
[...]
6: int8unsigned[stack(0)] = x10
[...]
4: x9 = int8signed[currentCodeLen + 0]   →   4: x9 = x16
3: x7 = x9 + -8                           →   3: x7 = x15

```

The memory store at point 6 was analyzed as addressing offset 0 of the stack block, which cannot alias the global block addressed at point 16. Therefore, when we read that same location at point 4, CSE knows that the result value is the same as that computed at point 16, and therefore reuses the result `x16` of the load at 16. In turn, CSE simplifies the add at point 3, as it knows that the same computation already took place at point 15.

## 8 Conclusions and Perspectives

An easy simplification that could reduce the cost of alias analysis is to restrict ourselves to points-to sets that are singletons, i.e. a single abstract location instead of a set of abstract locations. The experimental evaluation shows that points-to sets of cardinality 2 or more rarely appear, owing to our use of widening during fixpoint iteration. Moreover, those sets could be collapsed in a single abstract location at little loss of information just by adding a few extra points in the lattice of abstract locations depicted in Fig. 1.

Further improvements in analysis speed appear to require the use of more sophisticated, graph-based data structures, such as the alias graphs used by Larus and Hilfinger [10] and Steensgaard [16], among other authors. It is a challenge to implement and reason upon these data structures in a purely functional setting such as Coq. However, we could circumvent this difficulty by performing validation *a posteriori* in the style of Besson *et al.* [5]: an untrusted alias analysis, implemented in Caml using sophisticated data structures, produces a tentative map from program points to  $(\hat{R}, \hat{M})$  abstract states; a validator, written and proved correct in Coq, then checks that this tentative map is indeed a post-fixpoint of the dataflow inequations corresponding to our transfer function.

Concerning analysis precision, a first improvement would be to perform *strong updates* when the location stored into is uniquely known at analysis time, e.g. when the set of accessed abstract locations is a singleton of the form  $\{(\text{Stack}, i)\}$  or  $\{(\text{Globals}(\text{Just } id), i)\}$ . In this case, the contents of the memory map for this location can be replaced by the points-to set of the right-hand side of the `store`, instead of being merged with this points-to set.

Another direction is to analyze the offset parts of pointer values more precisely. The flat lattice of integers that we currently use to track offset values could be replaced by integer intervals. More generally, the analysis could be parameterized over an arbitrary abstract domain of integers.

The next major improvement in precision would be to make the analysis interprocedural. Conceptually, the modifications to the abstract interpretation

framework are minimal, namely introducing  $\text{Stack}(f)$  and  $\text{Allocs}(f, p)$  abstract blocks that are indexed by the name of the function  $f$  where the corresponding stack allocation or dynamic allocation occurs. However, the fixpoint iteration strategy must be changed: in particular, for a call to a function pointer, the points-to set of the function pointer is used to determine the possible successors of the call. In addition, issues of algorithmic efficiency and sparse data structures become much more acute in the interprocedural case.

## References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press (1998)
3. Bertot, Y.: Structural Abstract Interpretation: A Formal Study Using Coq. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) LerNet 2008. LNCS, vol. 5520, pp. 153–194. Springer, Heidelberg (2009)
4. Besson, F., Cachera, D., Jensen, T.P., Pichardie, D.: Certified Static Analysis by Abstract Interpretation. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 223–257. Springer, Heidelberg (2009)
5. Besson, F., Jensen, T., Pichardie, D.: Proof-carrying code from certified abstract interpretation to fixpoint compression. *Theoretical Computer Science* 364(3), 273–291 (2006)
6. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
7. Dabrowski, F., Pichardie, D.: A Certified Data Race Analysis for a Java-like Language. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 212–227. Springer, Heidelberg (2009)
8. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Program Analysis For Software Tools and Engineering (PASTE 2001), pp. 54–61. ACM (2001)
9. Kildall, G.A.: A unified approach to global program optimization. In: 1st Symposium Principles of Programming Languages, pp. 194–206. ACM Press, New York (1973)
10. Larus, J.R., Hilfinger, P.N.: Detecting conflicts between structure accesses. In: Programming Language Design and Implementation (PLDI 1988), pp. 21–34. ACM Press, New York (1988)
11. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* 52(7), 107–115 (2009)
12. Leroy, X.: A formally verified compiler back-end. *J. Automated Reasoning* 43(4), 363–446 (2009)
13. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Research report RR-7987, INRIA (June 2012)
14. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Automated Reasoning* 41(1) (2008)
15. Nipkow, T.: Abstract Interpretation of Annotated Commands. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 116–132. Springer, Heidelberg (2012)
16. Steensgaard, B.: Points-to analysis in almost linear time. In: 23rd Symp. Principles of Programming Languages (POPL 1996), pp. 32–41. ACM (1996)

# Mechanized Verification of Computing Dominators for Formalizing Compilers

Jianzhou Zhao and Steve Zdancewic

University of Pennsylvania  
{[jianzhou](mailto:jianzhou@cis.upenn.edu), [stevez](mailto:stevez@cis.upenn.edu)}@cis.upenn.edu

**Abstract.** One prerequisite to the formal verification of modern compilers is to formalize computing dominators, which enable SSA forms, advanced optimizations, and analysis. This paper provides an abstract specification of dominance analysis that is sufficient for formalizing modern compilers; it describes a certified implementation and instance of the specification that is simple to design and reason about, and also reasonably efficient. The paper also presents applications of dominance analysis: an SSA-form type checker, verifying SSA-based optimizations, and constructing dominator trees. This development is a part of the Vellvm project. All proofs and implementation have been carried out in Coq.

## 1 Introduction

Compilers are not always correct due to the complexity of language semantics and transformation algorithms, the trade-offs between compilation speed and verifiability, *etc.* Bugs in compilers can undermine the source-level verification efforts (such as type systems, static analysis, and formal proofs), and produce target programs with different meaning from source programs. The CompCert project [12] first implemented a realistic and mechanically verified compiler with classic intermediate representations in the Coq proof assistant. The CompCert compiler generates compact and efficient assembly code for a large fragment of the C language, and is proved to be more robust than non-verified compilers.

Recently researchers started to formalize and verify modern compilers in the Vellvm project [14] and in the CompCertSSA project [3]. One crucial component of modern compilers, such as LLVM and GCC, is computing *dominators*—on a control-flow-graph, a node  $l_1$  dominates a node  $l_2$  if all paths from the entry to  $l_2$  must go through  $l_1$  [2]. Dominance analysis allows compilers to represent programs in the SSA form [6] (which enables many advanced SSA-based optimizations), optimize loops, analyze memory dependency, and parallelize code automatically, *etc.* Therefore, one prerequisite to the formal verification of modern compilers is to formalize computing dominators.

In this paper, we present the formalization of dominance analysis used in the Vellvm project. To the best of our knowledge, this is the first mechanized verification of dominator computation for LLVM. Although the CompCertSSA project [3] also formalized dominance analysis to prove the correctness of a global value numbering optimization, our results are more general: beyond soundness,

we establish completeness and related metatheory results that can be used in other applications. Because different styles of formalization may also affect the cost of proof engineering, we also discuss some tradeoffs in the choices of formalization. In this work, we evaluate our formalism by applying it to several applications in Vellvm.

To simplify the formal development, we describe the work in the context of Vminus, which is a simpler subset of the full LLVM SSA IR formalized in Vellvm, that still captures the essence of dominance analysis. Our Coq development formalizes all the claims of the paper for the full Vellvm<sup>1</sup>. Following LLVM, we distinguish dominators at the block level and at the instruction level. Given the former one, we can easily compute the latter one. Therefore, we will focus on the block-level analysis, and discuss the instruction-level analysis only briefly.

We present the following contributions. Section 2 gives a specification of computing dominators at the block level. We instantiate the specification by two algorithms. Section 3 shows the standard dominance analysis [1] (AC). Section 4 presents an extension of AC [5] (CHK) that is easy to implement and verify, but still fast. We verify the correctness of both algorithms. Section 3.1 provides a verified depth first search algorithm. Then, Section 5 extends the dominance analysis to the instruction level, and present several applications used in the Vellvm project: a type checker for SSA, verifying SSA-based optimizations, and constructing dominator trees. Section 6 evaluates performance of the algorithms, and shows that in practice CHK runs nearly as fast as the LLVM’s algorithm.

## 2 The Specification of Computing Dominators

This section first defines dominators in term of the syntax of Vminus, then gives an abstract and succinct specification of algorithms that compute dominators.

**Syntax of Vminus.** Figure 1 gives the syntax of Vminus, focusing on the syntax of Vminus at the block level. Section 5 will revisit the rest of the syntax. All code in Vminus resides in top-level functions, whose bodies are composed of blocks  $b$ . Here,  $\bar{b}$  denotes a list of blocks; we use similar notation for other lists throughout the paper. As in classic compiler representations, a basic block consists of a label  $l$ , a series of instructions *insn* followed by a terminator *tmn* (**br** and **ret**) that branches to another block or returns from the function. In the following, we also use the label of a block to denote the block.

The set of blocks making up the top-level function  $f$  constitutes a control-flow graph (CFG)  $G = (e, succs)$  where  $e$  is the entry point (the first block) of  $f$ ; *succs* maps each label to a list of its successors. On a CFG, we use  $G \models l_1 \rightarrow^* l_2$  to denote a path  $\rho$  from  $l_1$  to  $l_2$ , and  $l \in \rho$  to denote that  $l$  is in the path  $\rho$ . By **wf f**, we require that a well-formed function must contain an entry point that cannot be reached from other blocks, all terminators can only branch to blocks within  $f$ , and that all labels in  $f$  are unique. In this paper, we consider only well-formed functions to streamline the presentation.

<sup>1</sup> Available at <http://www.cis.upenn.edu/~jianzhou/Vellvm/dominance>

Types	$typ ::= \mathbf{int}$	Instructions	$insn ::= \phi \mid c \mid tmn$
Constants	$cst ::= Int$	Phi Nodes	$\phi ::= r = \mathbf{phi} \, typ \, \overline{[val_j, l_j]^j}$
Values	$val ::= r \mid cst$	Commands	$c ::= r := val_1 \, bop \, val_2$
Blocks	$b ::= l \, \overline{\phi} \, \overline{tmn}$	Terminators	$tmn ::= \mathbf{br} \, l \mid \mathbf{br} \, val \, l_1 \, l_2 \mid \mathbf{ret} \, typ \, val$
Functions	$f ::= \mathbf{fun} \, \{\overline{b}\}$	Non- $\phi$ s	$\hat{\phi} ::= c \mid tmn$

Fig. 1. Syntax of Vminus

**Definition 1 (Domination (Block-level)).** Given  $G$  with an entry  $e$ ,

- A block  $l$  is **reachable**, written  $G \rightarrow^* l$ , if there exists a path  $G \models e \rightarrow^* l$ .
- A block  $l_1$  **dominates** a block  $l_2$ , written  $G \models l_1 \gg= l_2$ , if for every path  $\rho$  from  $e$  to  $l_2$ ,  $l_1 \in \rho$ .
- A block  $l_1$  **strictly dominates** a block  $l_2$ , written  $G \models l_1 \gg l_2$ , if for every path  $\rho$  from  $e$  to  $l_2$ ,  $l_1 \neq l_2 \wedge l_1 \in \rho$ .

Because the dominance relations of a function at the block level and in its CFG are equivalent, in the following we do not distinguish  $f$  and  $G$ . The following consequence of the definitions are useful to define the specification of computing dominators. For all labels in  $G$ ,  $\gg=$  and  $\gg$  are transitive.

**Lemma 1**

- If  $G \models l_1 \gg= l_2$  and  $G \models l_2 \gg= l_3$ , then  $G \models l_1 \gg= l_3$ .
- If  $G \models l_1 \gg l_2$  and  $G \models l_2 \gg l_3$ , then  $G \models l_1 \gg l_3$ .

However, because there is no path from the entry to unreachable labels,  $\gg=$  and  $\gg$  relate every label to any unreachable labels.

**Lemma 2.** If  $\neg(G \rightarrow^* l_2)$ , then  $G \models l_1 \gg= l_2$  and  $G \models l_1 \gg l_2$ .

If we only consider the reachable labels in  $V$ ,  $\gg$  is acyclic.

**Lemma 3 ( $\gg$  is acyclic).** If  $G \rightarrow^* l$ , then  $\neg G \models l \gg l$ .

Moreover, all labels that strictly dominate a reachable label are ordered.

**Lemma 4 ( $\gg$  is ordered).** If  $G \rightarrow^* l_3$ ,  $l_1 \neq l_2$ ,  $G \models l_1 \gg l_3$  and  $G \models l_2 \gg l_3$ , then  $G \models l_1 \gg l_2 \vee G \models l_2 \gg l_1$ .

## 2.1 Specification

**Coq Notations.** We use  $\{\}$  to denote an empty set; use  $\{+\}$ ,  $\{<=\}$ , ‘In’,  $\{\vee\}$  and  $\{\wedge\}$  to denote set addition, inclusion, membership, union and intersection respectively. Our developments reuse the basic tree and map data structures implemented in the CompCert project [12]: `ATree.t` and `PTree.t` are trees with keys of type  $l$  and positive respectively; `PMap.t` is a map with keys of type positive. We use  $[\ ]$  to denote tree and map lookup. `succs` are defined by trees.  $[\ ]$  returns an empty list when a searched-for key in `succs` does not exist.  $[\mathbf{x}]$  is a list with one element  $\mathbf{x}$ .

```

Module Type ALGDOM.
  Parameter sdom: f →l →set l.
  Definition dom f l1 := l1 {+} sdom f l1.
  Axiom entry_sound: forall f e, entry f = Some e →sdom f e = {}.
  Axiom successors_sound: forall f l1 l2,
    In l1 (succs f)[l2] →sdom f l1 {<=} dom f l2.
  Axiom complete: forall f l1 l2,
    wf f →f |= l1 >> l2 →l1 'In' (sdom f l2).
End ALGDOM.
Module AlgDom_Properties(AD: ALGDOM).
  Lemma sound: forall f l1 l2,
    wf f →l1 'In' (AD.sdom f l2) →f |= l1 >> l2.
  (*****
  (* Properties: conversion, transitivity, acyclicity, ordering and ... *)
  (*****
End AlgDom_Properties.

```

Fig. 2. The specification of algorithms that find dominators

Figure 2 gives an abstract specification of algorithms that compute dominators using a Coq module interface `ALGDOM`. First of all, `sdom` defines the signature of a dominance analysis algorithm: given a function  $f$  and a label  $l_1$ ,  $(\text{sdom } f \ l_1)$  returns the set of strict dominators of  $l_1$  in  $f$ ; `dom` defines the set of dominators of  $l_1$  by adding  $l_1$  into  $l_1$ 's strict dominators.

To make the interface simple, `ALGDOM` only requires the basic properties that ensure that `sdom` is correct: it must be both `sound` and `complete` in terms of the declarative definitions (Definition 1). Given the correctness of `sdom`, the `AlgDom_Properties` module can ‘lift’ properties (conversion, transitivity, acyclicity, ordering, *etc.*) from the declarative definitions to the implementations of `sdom` and `dom`. Section 5 shows how clients of `ALGDOM` use the properties proven in `AlgDom_Properties` by examples.

`ALGDOM` requires completeness directly. Soundness can be proven by two more basic properties: `entry_sound` requires that the entry has no strict dominators; `successors_sound` requires that if  $l_1$  is a successor of  $l_2$ , then  $l_2$ 's dominators must include  $l_1$ 's strict dominators. Given an algorithm that establishes the two properties, `AlgDom_Properties` proves that the algorithm is sound by induction over any path from the entry to  $l_2$ .

## 2.2 Instantiations

In the literature, there is a long history of algorithms that find dominators, each making different trade-offs between efficiency and simplicity. Most of the industry compilers, such as LLVM and GCC, use the classic Lengauer-Tarjan algorithm [11] (LT) that has a complexity of  $O(E * \log(N))$  where  $N$  and  $E$  are the number of nodes and edges respectively, but is complicated to implement and reason about. The Allen-Cocke algorithm [1] (AC) based on iteration is easier to design, but suffers from a large asymptotic complexity. Moreover, LT explicitly creates dominator trees that provide convenient data structures for

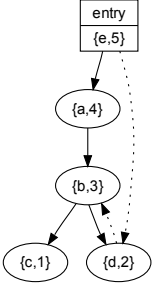


Fig. 3. Postorder

stk	visited	PO.l2p po
e[a d]	e	
e[d]; a[b]	e a	
e[d]; a[]; b[c d]	e a b	
e[d]; a[]; b[d]; c[]	e a b c	(c,1)
e[d]; a[]; b[]; d[b]	e a b c d	(c,1)
e[d]; a[]; b[]; d[]	e a b c d	(c,1); (d,2)
e[d]; a[]; b[];	e a b c d	(c,1); (d,2); (b,3)
e[d]; a[];	e a b c d	(c,1); (d,2); (b,3); (a,4)
e[]	e a b c d	(c,1); (d,2); (b,3); (a,4); (e,5)

Fig. 4. The DFS execution sequence

compilers whereas AC needs an additional tree construction algorithm with more overhead. The Cooper-Harvey-Kennedy algorithm [5] (CHK), extended from AC with careful engineering, runs nearly as fast as LT in common cases [5,8], but is still simple to implement and reason about. Moreover, CHK generates dominator trees implicitly, and provides a faster tree construction algorithm.

Because CHK gives a relatively good trade-off between verifiability and efficiency, we present CHK as an instance of ALGDOM. In the following sections, we first review the AC algorithm, and then study its extension CHK.

### 3 The Allen-Cocke Algorithm

The Allen-Cocke algorithm (AC) is an instance of the forward worklist-based Kildall’s algorithm [10] that visits nodes in reverse postorder (PO) [9] (in which AC converges faster). At the high-level, our Coq implementation of AC works in three steps: 1) calculate the PO of a CFG by depth-first-search (DFS); 2) compute strict dominators for PO-numbered nodes in Kildall; 3) finally relate the analysis results to the original nodes. We omit the 3rd step’s proofs here.

This section first presents a verified DFS algorithm that computes PO, then reviews Kildall’s algorithm as implemented in the CompCert project [12], and finally it studies the implementation and metatheory of AC.

#### 3.1 DFS: PO-Numbering

DFS starts at the entry, visits nodes as deep as possible along each path, and backtracks when all deep nodes are visited. DFS generates PO by numbering a node after all its children are numbered. Figure 3 gives a PO-numbered CFG. In the CFG, we represent the depth-first-search (DFS) tree edges by solid arrows, and non-tree edges by dotted arrows. We draw the entry node in a box, and other nodes in circles. Each node is labeled by a pair with its original label name on the left, and its PO number on the right. Because DFS only visits reachable nodes, the PO numbers of unreachable nodes are represented by ‘\_’.

Figure 5 shows the data structures and auxiliary functions used by a typical DFS algorithm that maintains four components to compute PO. `PostOrder`



```

Record PostOrder := mkPO { PO_cnt: positive; PO_l2p: LTree.t positive }.
Record Frame := mkFr { Fr_name: l; Fr_scs: list l }.
Definition dfs_F_type : Type := forall (succs: LTree.t (list l))
  (visited: LTree.t unit) (po:PostOrder) (stk: list Frame), PostOrder.
Definition dfs_F (f: dfs_F_type) (succs: LTree.t (list l))
  (visited: LTree.t unit) (po:PostOrder) (stk: list Frame): PostOrder :=
  match find_next succs visited po stk with
  | inr po' =>po'
  | inl (next, visited', po', stk') =>f succs visited' po' stk'
end.

```

Fig. 5. The DFS algorithm

takes the next available PO number and a map from nodes to their PO numbers with type `positive`. `succs` maps a node to its successors. To facilitate reasoning about DFS, we represent the recursive information of DFS explicitly by a list of `Frame` records that each contains a node `Fr_name` and its unprocessed successors `Fr_scs`. To prevent the search from revisiting nodes, the DFS algorithm uses `visited` to record visited nodes. `dfs_F` defines one recursive step of DFS.

Figure 4 gives a DFS execution sequence (by running `dfs_F` until all nodes are visited) of the CFG in Figure 3. We use  $l[l_1 \dots l_n]$  to denote a frame with the node  $l$  and its unprocessed successors  $l_1$  to  $l_n$ ;  $(l, p)$  to denote a node  $l$  and its PO  $p$ . Initially the DFS adds the entry and its successors to the stack. At each recursive step, `find_next` finds the next available node that is the unvisited node in the `Fr_scs` of the latest node  $l'$  of the stack. If the next available node exists, the DFS pushes the node with its successors to the stack, and makes the node to be visited. `find_next` pops all nodes in front of  $l'$ , and gives them PO numbers. If `find_next` fails to find available nodes, the DFS stops.

We can see that the straightforward algorithm is not a structural recursion. To implement the algorithm in Coq, we must show that it terminates. Although in Coq we can implement the algorithm by well-founded recursion, such designs are hard to reason about [4]. One possible alternative is implementing DFS with a ‘strong’ dependent type to specify the properties that we need to reason about DFS. However, this design is not modular because when the type of DFS is not strong enough—for example, if we need a new lemma about DFS—we must extend or redesign its implementation by adding new invariants. Instead, following the ideas in Coq’Art [4], we implement DFS by iteration and prove its termination and inductive principle separately. By separating implementation and specification, the DFS design is modular, and easier to reason about.

Figure 6 presents our design. The top-level entry is `iter`, which needs a bounding step `n`, a fixpoint `F` and a default value `g`. `iter` only calls `g` when `n` reaches zero, and otherwise recursively calls one more iteration of `F`. If `F` is terminating, we can prove that there must exist a final value and a bound `n`, such that for any bound `k` that is greater than or equal to `n`, `iter` always stops and generates the same final value. In other words, `F` reaches a fixpoint in fewer than `n` steps. The proof of the existence of `n` is erasable; the computation part provides a terminating algorithm, not requiring the bound step at runtime.

```

Fixpoint iter (A:Type) (n:nat) (F:A→A) (g:A) : A :=
  match n with
  | 0 =>g
  | S p =>F (iter A p F g)
  end.
Definition wf_stk succs visited stk :=
  stk_in_succs succs stk ^incl visited succs
Program Fixpoint dfs_tmn succs visited po stk
  (Hp: wf_stk succs visited stk) {measure (size succs - size visited)}:
  { po':PostOrder | exists p:nat,
    forall k (Hlt: p < k) (g:dfs_F_type),
    iter _ k dfs_F g succs visited po stk = po' } :=
  match find_next succs visited po stk with
  | inr po' =>po'
  | inl (next, visited', po', stk') =>
    let _ := dfs_tmn succs visited' po' stk' _ in _
  end.
Program Definition dfs succs entry : PostOrder :=
fst (dfs_tmn succs empty (mkPO 1 empty) (mkFr entry [succs[entry]]) _).

```

Fig. 6. Termination of the DFS algorithm

In Figure 6, `dfs_tmn` proves DFS termination, which is established by well-founded recursion over the number of unvisited nodes. This holds because each iteration the DFS visits more nodes. The invariant that the number of unvisited nodes decreases holds only for well-formed recursion states (`wf_stk`), which requires that all visited nodes and unprocessed nodes in frames are in the CFG.

To reason about `dfs`, we defined a well-founded inductive principle for `dfs` (See our code). With the inductive principle, we proved the following properties of DFS that are useful to establish the correctness of AC and CHK.

**Variable** (`succs`: `ATree.t (list l)`) (`entry`:`l`) (`po`:`PostOrder`).

**Hypothesis** `Hdfs`: `dfs succs entry = po`.

First of all, a non-entry node must have at least one predecessor that has a greater PO number than the node's. This is because 1) DFS must visit at least one predecessor of a node before visiting the node; 2) PO gives greater numbers to the nodes visited earlier:

**Lemma** `dfs_order`: `forall l1 p1, l1 <> entry →(PO_l2p po) [l1] = Some p1, exists l2, exists p2,`

`In l2 ((make_preds succs) [l1]) ^ (PO_l2p po) [l2] = Some p2 ^ p2 > p1.`

*(\* Given succs, (make\_preds succs) computes predecessors of each node. \*)*

Second, a node is PO-numbered iff the node is reachable:

**Lemma** `dfs_reachable`:`forall l, (PO_l2p po) [l] <> None ↔ (entry, succs) →* l.`

Moreover, different nodes do not have the same PO number.

**Lemma** `dfs_inj`: `forall l1 l2 p,`

`(PO_l2p po) [l2] = Some p →(PO_l2p po) [l1] = Some p →l1 = l2.`

```

Module Kildall (NS: PNODE_SET) (L: LATTICE). Section Kildall.
Variable succs: PTree.t (list positive).
Variable transf : positive →L.t →L.t.
Variable inits: list (positive * L.t).
Record state : Type := mkst { sin: PMap.t L.t; swrk: NS.t }.
Definition start_st := mkst (start_state_in inits) (NS.init succs).
Definition propagate_succ (out: L.t) (s: state) (n: positive) :=
  let oldl := s.(sin)[n] in
  let newl := L.lub oldl out in
  if L.eq newl oldl
  then mkst (PMap.set n newl s.(sin)) (NS.add n s.(swrk)) else s.
Definition step (s: state): PMap.t L.t + state :=
  match NS.pick s.(swrk) with
  | None ⇒inl s.(sin)
  | Some(n, rem) ⇒inr (fold_left
                        (propagate_succ (transf n s.(sin)[n]))
                        succs[n] (mkst s.(sin) rem))

  end.
Variable num : positive.
Definition fixpoint : option (PMap.t L.t) := Iter.iter step num start_st.
End Kildall. End Kildall.

```

Fig. 7. Kildall’s algorithm

### 3.2 Kildall’s Algorithm

Figure 7 summarizes the Kildall module used in the CompCert project. The module is parameterized by the following components: `NS` that provides the order to process nodes, and a lattice `L` that defines `top`, `bot`, equality (`eq`), least upper bound (`lub`) and order (`ge`) of the abstract domain of an analysis; `succs` that is a tree that maps each node to its successors; `transf` that is the transfer function of Kildall analysis; `inits` that initializes the analysis. Given the inputs, `state` records the iteration states that include `sin`, which records analysis states for each node, and a work list `swrk` containing nodes to process.

The `fixpoint` implements iterations by `Iter.iter`—bounded recursion with a maximal step number (`num`) [4]. `Iter.iter` is partial if an analysis does not stop after the maximal number of steps. A monotone analysis must reach its fixpoint after a finite number of steps. Therefore, we can always pick a large enough number of steps for a monotone analysis.

Initially Kildall’s algorithm calls `start_st` to initialize iteration states. Nodes not in `inits` are initialized to be the bottom of `L`. Then `start_st` adds all nodes into the worklist and starts loops. `step` defines the loop body. At `step`, Kildall’s algorithm checks if there are still unprocessed nodes in the worklist. If the worklist is empty, the algorithm stops. Otherwise, `step` picks a node from the worklist in term of the order provided by `NS`, and then propagates its information (computed by `transf`) to all the node’s successors by `propagate_succ`. In `propagate_succ`, the new value of a successor is `L.lub` of its old value and

the propagated value from its predecessor. The algorithm only adds a successor into the worklist when its value is changed.

Kildall's algorithm satisfies the following properties:

**Variable** `res`: PMap.t L.t.

**Hypothesis** `Hfix`: `fixpoint = Some res`.

First of all, the worklist contains nodes that have unstable successors in the current state. Formally, each state `st` preserves the following invariant:

**forall** `n`, `NS.In n st.(swrk) ∨`  
`(forall s, In s (succs[n]) → L.ge st.(sin)[s] (transf n st.(sin)[n]))`.

Each iteration may only remove the picked node `n` from the worklist. If none of `n`'s successors' values are changed, no matter whether `n` belongs to its successors, `n` won't be added back to the worklist. Therefore, the above invariant holds. This invariant implies that when the analysis stops, all nodes hold the in-equations:

**Lemma** `fixpoint_solution`: **forall** `s`,  
`In s (succs[n]) → L.ge res[s] (transf n res[n])`.

The second property of Kildall's algorithm is *monotonicity*. At each iteration, the value of a successor of the picked node can only be updated from `old1` to `new1`. Because `new1` is the least upper bound of `old1` and `out`, `new1` is greater than or equal to `old1`. Therefore, iteration states are always monotonic:

**Lemma** `fixpoint_mono`: `incr (start_state_in inits) res`.

where `incr` is a pointwise lift of `L.ge` for corresponding nodes. With monotonicity, we proved that Kildall's algorithm must terminate (See our code).

### 3.3 The AC Algorithm

AC instantiates `Kildall` with `PN` that picks nodes in reverse PO (by picking the maximal nodes from the worklist), and `LDoms` that defines the lattice of AC. Dominance analysis computes a set of strict dominators for each node. We represent the domain of `LDoms` by `option (set 1)`. The `top` and `bot` of `LDoms` are `Some nil` and `None` respectively. The least upper bound, order and equality of `LDoms` are lifted from set intersection, set inclusion, and set equality to `option`: `None` is smaller than `Some x` for any `x`. This design leads to better performance by providing shortcuts for operations on `None`. Note that using `None` as `bot` does not make the height of `LDoms` to be infinite, because any non-`bot` element can only contain nodes in the CFG, and the height of `LDoms` is  $N$ .

AC uses the following transfer function and initialization:

**Definition** `transf l1 input := l1 {+} input`.

**Definition** `inits := [(e, LDoms.top)]`.

Initially AC sets the strict dominators of the entry to be empty, and other nodes' strict dominators to be all labels in the function. The algorithm will iteratively remove non-strict-dominators from the sets until the conditions below hold (by Lemma `fixpoint_mono` and Lemma `fixpoint_solution`):

**(forall** `s`, `In s (succs[n]) →`  
`L.ge (st.(sin))[s] (n{+}(st.(sin))[n]) ∧ (st.(sin))[e] = {}`).

which proves that AC satisfies `entry_sound` and `successors_sound`.

To show that the algorithm is complete, it is sufficient to show that each iteration state `st` preserves the following invariant:

```
forall n1 n2, ~n1 'In' st.(sin)[n2] →~(e, succs) |= n1 >> n2.
```

In other words, AC only removes non-strict dominators. Initially, AC sets the entry's strict dominators to be empty. Because in a well-formed CFG, the entry has no predecessors, the invariant holds at the very beginning. At each iteration, suppose that we pick a node `n`, and updates one of its successors `s`. Consider a node `n'` not in `LDoms.lub st.(sin)[s] (n {+} st.(sin)[n])`. If `n'` is not in `LDoms.lub st.(sin)[s]`, then `n'` does not strictly dominate `s` because `st` holds the invariant. If `n'` is not in `(n {+} st.(sin)[n])`, then `n'` does not strictly dominate `n` because `st` holds the invariant. Appending the path from the entry to `n` that bypasses `n'` with the edge from `n` to `s` leads to a path from the entry to `s` that bypasses `n'`. Therefore, `n'` does not strictly dominate `s`, either.

## 4 Extension: The Cooper-Harvey-Kennedy Algorithm

The CHK algorithm is based on the following observation: when AC processes nodes in a reversed post-order (PO), if we represent the set of strict dominators in a list, and always add a newly discovered strict dominator at the head of the list (on the left in Figure 9), the list must be sorted by PO. Figure 9 shows the execution of the algorithm for the CFG in Figure 3.

Because lists of strict dominators are always sorted, we can implement the set intersection (`lub`) and the set comparison (`eq`) of two sorted lists by traversing the two lists only once. Moreover, the algorithm only calls `eq` after `lub`. Therefore, we can group `lub` and `eq` into `LDoms.lub` together. The following defines a `merge` function used by `LDoms.lub` that intersects two sorted lists and returns whether the final result is equal to the left one:

```
Program Fixpoint merge (l1 l2: list positive) (acc: list positive * bool)
  {measure (length l1 + length l2)}: (list positive * bool) :=
  let '(r1, changed) := acc in
  match l1, l2 with
  | p1::l1', p2::l2' =>
    match (Pcompare p1 p2 Eq) with
    | Eq => merge l1' l2' (p1::r1, changed)
    | Lt => merge l1' l2 (r1, true)
    | Gt => merge l1 l2' (r1, changed)
    end
  | nil, _ => acc
  | _::_, nil => (r1, true)
  end.
(* (Pcompare p1 p2 Eq) returns whether p1 = p2, p1 < p2 or p1 > p2. *)
```

### 4.1 Correctness

To show that CHK is still correct, it is sufficient to show that all lists are well-sorted at each iteration, which ensures that the above `merge` correctly imple-

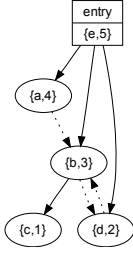


Fig. 8. Dominator Trees

Nodes	sin								
5	∅	∅	∅	∅	∅	∅	∅	∅	∅
4	.	[5]	[5]	[5]	[5]	[5]	[5]	[5]	[5]
3	.	.	[45]	[45]	[45]	[5]	[5]	[5]	[5]
2	.	.	.	[345]	[345]	[345]	[35]	[35]	[35]
1	.	[5]	[5]	[5]	[5]	[5]	[5]	[5]	[5]
swrk	[54321]	[4321]	[321]	[21]	[1]	[3]	[21]	[1]	∅

Fig. 9. The execution of CHK

ments intersection and comparison. First, if a node with number  $n$  still maps to `bot`, the worklist must contain one of its predecessors that has a greater number.

```
forall n, in_cfg n succs → (st.(sin))[n] = None →
exists p, In p ((make_preds succs) [n]) ∧ p > n ∧ PN.In p st.(st_wrk).
(* in_cfg checks if a node is in CFG. *)
```

This invariant holds in the beginning because all nodes are in the worklist. At each iteration, the invariant implies that the picked node  $n$  with the maximal number in `st.(st_wrk)` is not `bot`. Suppose it is `bot`, there cannot be any node with greater number in the worklist. This property ensures that after each iteration, the successors of  $n$  cannot be `bot`, and that the new nodes added into the worklist cannot be `bot`, because they must be those successors. Therefore, the predecessors of the remaining `bot` nodes still in the worklist cannot be  $n$ . Since only  $n$  is removed, the rest of the `bot` nodes still hold the above invariant.

In the algorithm, a node's value is changed from `bot` to non-`bot` when one of its non-`bot` predecessors is processed. With the above invariant, we know that the predecessor must be of larger number. Once a node turns to be non-`bot`, no new elements will be added in its set. Therefore, this implies that, at each iteration, if the value of a node is not `bot`, then all its candidate strict dominators must be larger than the node:

```
forall n sdms, (st.(sin))[n] = Some sdms → forall (Plt n) sdms.
(* Plt is the less-than of positive. *)
```

Moreover, a node  $n$  is considered as a candidate of strict dominators originally by `tranf` that always cons  $n$  at the head of `(st.(sin))[n]`. Therefore, we proved that the non-`bot` value of a node is always sorted:

```
forall n sdms, (st.(sin))[n] = Some sdms → Sorted Plt (n::sdms).
```

## 5 Applications

### 5.1 Type Checker

The first application is the type checker of Vminus. The Vminus language in Figure 11 is in SSA form [6] in which each variable may be defined only once, statically, and each use of the variable must be dominated by its definition with respect to the control-flow graph of the containing function. To maintain these invariants in the presence of branches and loops, SSA form uses  $\phi$ -instructions to

merge definitions from different incoming paths. As usual in SSA representation, the  $\phi$  nodes join together values from a list of predecessor blocks of the control-flow graph—each  $\phi$  node takes a list of (value, label) pairs that indicates the value chosen when control transfers from a predecessor block with the associated label.

To check that a program is in SSA form, we need to extend domination relations from the block-level to the instruction-level. Instruction positions are denoted by program counters  $pc$ . We write  $f[pc] = [insn]$  if  $insn$  is at  $pc$  of  $f$ .

### Definition 2 (Domination (Instruction-level))

- $val$  **uses**  $r \triangleq val = r$ .
- $insn$  **uses**  $r \triangleq \exists val. val$  **uses**  $r \wedge val$  is an operand of  $insn$ .
- A variable  $r$  is defined at a program counter  $pc$  of  $f$ , written  $f$  **defines**  $r @ pc$ , if  $f[pc] = [insn]$  and  $r$  is the left-hand side of  $insn$ .
- In function  $f$ ,  $pc_1$  **strictly dominates**  $pc_2$ , written  $f \models pc_1 \gg pc_2$ , if  $pc_1$  and  $pc_2$  are at distinct blocks  $l_1$  and  $l_2$  respectively and  $f \models l_1 \gg l_2$ ; or if  $pc_1$  and  $pc_2$  are in the same block, and  $pc_1$  appears earlier than  $pc_2$ .
- $\mathbf{sdom}_f(pc)$  is the set of variables whose definitions strictly dominate  $pc$ :  $\{r \mid f$  **defines**  $r @ pc'$  and  $f \models pc' \gg pc\}$

Then we check if a program is of SSA form with the following rules:

$$\frac{\forall r. (\hat{\phi} \text{ uses } r \implies r \in \mathbf{sdom}_f(pc))}{f \vdash \hat{\phi} @ pc} \quad \frac{\forall r_j. (\overline{val_j \text{ uses } r_j \implies r_j \in \mathbf{sdom}_f(l_j.t)})^j}{f \vdash r = \mathbf{phi\ typ} [\overline{val_j, l_j}]^j}$$

The left rule ensures that a non- $\phi$ -instruction ( $c$  or  $tmn$ ) can only use the definitions in the scope of  $\mathbf{sdom}_f(pc)$ ; the right rule ensures that in  $\phi$ , an incoming value must use the definition that strictly dominates the end of the corresponding incoming block where  $l.t$  is the program counter at the end of  $l$ . Please refer to [14] for the type safety proofs of Vminus.

## 5.2 SSA-Based Optimizations

The SSA form is good for implementing optimizations because the SSA invariants make def/use information of variables explicit, enforcing fewer mutable states [2]. An SSA-based transformation is correct if it preserves the semantics of the original program and its transformed program is still in SSA. Here, we briefly show how to reason about well-formedness-preservation by examples.

First, we proved that the strict domination relation at the instruction level still satisfies transitivity and acyclicity.

### Lemma 5

- If  $f \models pc_1 \gg pc_2$  and  $f \models pc_2 \gg pc_3$ , then  $f \models pc_1 \gg pc_3$ .
- If  $pc$  is in a reachable block, then  $\neg f \models pc \gg pc$ .

Consider the following typical SSA-based optimization:

Original	Transformed
$e : \dots$	$e : \dots$
$@pc_1$	$r_4 := r_1 * r_2$
$\mathbf{br} r_0 l_1 l_2$	$\mathbf{br} r_0 l_1 l_2$
$l_1 : r_3 = \mathbf{phi int}[0, e][r_5, l_1]$	$l_1 : r_3 = \mathbf{phi int}[0, e][r_5, l_1]$
$@pc_2 r_4 := r_1 * r_2$	
$r_5 := r_3 + r_4$	$r_5 := r_3 + r_4$
$r_6 := r_5 \geq 100$	$r_6 := r_5 \geq 100$
$\mathbf{br} r_6 l_1 l_2$	$\mathbf{br} r_6 l_1 l_2$
$l_2 : r_7 = \mathbf{phi int}[0, e][r_5, l_2]$	$l_2 : r_7 = \mathbf{phi int}[0, e][r_5, l_1]$
$@pc_3 r_8 := r_1 * r_2$	
$r_9 := r_8 + r_7$	$r_9 := r_4 + r_7$

In the original program,  $r_1 * r_2$  is a partial common expression for the definitions of  $r_4$  and  $r_8$ , because there is no domination relation between  $r_4$  and  $r_8$ . Therefore, eliminating the common expression directly is not correct.

We might transform this program in three steps. First, we move the instruction  $r_4 := r_1 * r_2$  from  $l_1$  to the end of  $e$ . Because  $e$  strictly dominates  $l_1$ , we have  $f \models pc_1 \gg pc_2$  where  $pc_1$  is exactly before  $e$ .  $\mathbf{t}$ ;  $f$  **defines**  $r_4$  @  $pc_2$ . By Lemma 5, the definition of  $r_4$  at  $pc_1$  should still strictly dominate all its uses.

We have  $f \models pc_1 \gg pc_3$  where  $f$  **defines**  $r_8$  @  $pc_3$ , because  $e$  strictly dominates  $l_2$ . Then, we can safely replace all the uses of  $r_8$  by  $r_4$ , because the definition of  $r_4$  at  $pc_1$  dominates all the uses of  $r_8$  (by Lemma 5).

Finally, by Lemma 5, we know that  $r_4$  and  $r_8$  cannot be equal. Therefore, we can remove  $r_8$ , because there are no uses of  $r_8$  after the substitution. The final program after the transformations is shown on the right of the above example.

### 5.3 Constructing Dominator Trees

In practice, compilers construct dominator trees from dominators, and analyze or optimize programs by recursion on dominator trees.

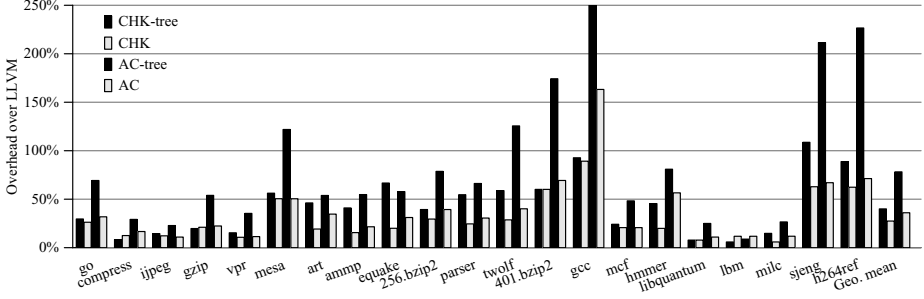
#### Definition 3

- A block  $l_1$  is an **immediate dominator** of a block  $l_2$ , written  $G \models l_1 \ggg l_2$ , if  $G \models l_1 \gg l_2$  and  $(\forall G \models l_3 \gg l_2, G \models l_3 \ggg l_1)$ .
- A tree is called a **dominator tree** of  $G$  if the tree has an edge from  $l$  to  $l'$  iff  $G \models l \ggg l'$ .

Figure 8 shows the dominator tree of a CFG, in which solid edges represent tree edges, and dotted edges represent non-tree but CFG edges. Formally, a dominator tree has the inductive **well-formed** property with which we can reason about recursion on dominator trees: given a tree node  $l$ , 1)  $l$  is reachable; 2)  $l$  is different from all labels in  $l$ 's descendants; 3) labels of  $l$ 's subtrees are disjointed; 4)  $l$  immediate-dominates its children; 5)  $l$ 's subtrees are well-formed.

Consider the final analysis results of CHK in Figure 9, we can see that for each node, its list of strict dominators exactly presents a path from root to the node on the dominator tree. Therefore, we can construct a dominator tree by





**Fig. 10.** Analysis overhead over LLVM’s dominance analysis for our extracted analysis

merging the paths. We proved that the algorithm correctly constructs a well-formed dominator tree (See our code). For the sake of space, we only present that each tree edge represents  $\ggg$  by showing that for any node  $l$  in the final state, the list of  $l$ ’s dominators must be sorted by  $\ggg$ .

We first show that the list is sorted by  $\ggg$ . Consider two adjacent nodes in the list,  $l_1$  and  $l_2$ , such that  $l_1 < l_2$ . Because of soundness,  $G \models l_1 \ggg l$  and  $G \models l_2 \ggg l$ . By Lemma 4,  $G \models l_2 \ggg l_1 \vee G \models l_1 \ggg l_2$ . Suppose  $G \models l_1 \ggg l_2$ , by completeness,  $l_1$  must be in the strict dominators computed for  $l_2$ , and therefore, be greater than  $l_2$ . This is a contradiction. Then, we prove that the list is sorted by  $\ggg$ . Suppose  $G \models l_3 \ggg l_1$ . By Lemma 1,  $G \models l_3 \ggg l$ . By completeness,  $l_3$  must be in the list. We have two cases: 1)  $l_3 \geq l_2$ : because the list is sorted by  $\ggg$ ,  $G \models l_3 \ggg l_2$ ; 2)  $l_3 \leq l_1$ : this is a contradiction by Lemma 3.

## 6 Performance Evaluation

We use Coq extraction to obtain a certified implementation of AC and CHK. We evaluate the performance of the resultant code on a 1.73 GHz Intel Core i7 processor with 8 GB memory running benchmarks selected from the SPEC CPU benchmark suite that consist of over 873k lines of C source code.

Figure 10 reports the analysis time overhead (smaller is better) over the LLVM dominance analysis (which uses LT) baseline. LT only generates dominator trees. Given a dominator tree, the strict dominators of a tree node are all the node’s ancestors. The second left bar of each group shows the overhead of CHK, which provides an average overhead of 27.45%. The right-most bar of each group is the overhead of AC, which provides 36.02% on average.

To study the asymptotic complexity, Figure 11 shows the result of graphs that elicit the worst-case behavior used in 8. On average, CHK is 86.59 times slower than LT. The ‘-’ indicates that the running time is too long to collect. For the testcases on which AC stops, AC is 226.14 times slower than LT.

The results of CHK match earlier experiments [8, 5]: in common cases, CHK runs nearly as fast as LT. For programs with reducible CFGs, a forward iteration analysis in reverse PO halts in no more than 6 passes [9], and most CFGs of the benchmarks are reducible. The worst-case tests contain huge irreducible CFGs.

Instance			Analysis Times (s)				
Name	Vertices	Edges	LT	CHK	CHK-tree	AC	AC-tree
idfsquad	6002	10000	0.08	10.54	24.87	-	-
ibfsquad	4001	6001	0.14	11.38	13.16	12.43	30.00
itworst	2553	5095	0.14	8.47	11.22	19.16	69.72
sncaworst	3998	3096	0.19	17.03	32.08	205.07	740.53

**Fig. 11.** Worst-case behavior

Different from these experiments, AC does not provide large overhead, because we use `None` to represent `bot`, which provides shortcuts for set operations.

As shown in Section 5.3, CHK computes dominator trees implicitly, while AC needs additional costs to create dominator trees. Figure 10 and Figure 11 also report the performance of the dominator tree construction. CHK-tree stands for the algorithm that first computes dominators by CHK, and then runs the tree construction defined in Section 5.3. AC-tree stands for the algorithm that first computes dominators by AC, sorts strict dominators for each node, and then runs the same tree construction. For common programs, on average, CHK-tree provides an overhead 40.00% over the baseline; AC-tree provides an overhead 78.20% over the baseline (gcc’s overhead is 361.23%). The additional overhead of AC-tree is from its sorting algorithm. For worst-case programs, on average, CHK-tree is 104.48 times slower than LT. For the testcases on which AC-tree stops, on average, AC-tree is 738.24 times slower than LT.

These results indicate that CHK makes a good trade-off between simplicity and efficiency.

## 7 Related Work

**Machine-Checked Formalizations.** The Vellvm project [14] uses dominance analysis to design a type checker of LLVM bitcode in SSA form. This paper extends and generalizes the implementation and metatheory in the Vellvm project. The CompCertSSA project [3] improves the CompCert compiler by creating a verified SSA-based middle-end. They also formalize the AC algorithm to validate SSA construction and GVN passes, and prove the soundness of AC. We implement both AC and CHK—an extension of AC in a generic way, and prove they are both sound and complete. We also provide the corresponding dominator tree constructions, and evaluate performance.

**Informal Formalizations.** Georgiadis and Tarjan [7] propose an almost linear-time algorithm that validates if a tree is a dominator tree of a CFG. Although the algorithm is fast, it is nearly as complicated as the LT algorithm, and it requires a substantial amount of graph theory. Ramalingam [13] proposes another dominator tree validation algorithm by reducing validating dominator trees to validating loop structures. However, in practice, most of modern loop identification algorithms used in LLVM and GCC are based on dominance analysis to find loop headers and bodies.

## 8 Conclusion

This paper provided an abstract specification of dominance analysis that is crucial for compiler design/verification and program analysis. We implemented and certified an instance of the specification that has a good trade-off between efficiency and simplicity. We also presented several applications of the analysis: a type checker for the SSA form; verifying SSA-based optimizations; and constructing dominator trees. This development is a part of the Vellvm project. However, our work might be used in other compiler verification projects [3].

**Acknowledgments.** We thank Santosh Nagarakatte and Milo Martin whose valuable discussions and technical input helped us carry out this research. This research was sponsored in part by NSF grant CCF-1065116. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

## References

1. Allen, F.E., Cocke, J.: Graph theoretic constructs for program control flow analysis. Technical report, IBM T.J. Watson Research Center (1972)
2. Appel, A.W.: Modern Compiler Implementation in C: Basic Techniques. Cambridge University Press (1997)
3. Barthe, G., Demange, D., Pichardie, D.: A Formally Verified SSA-Based Middle-End. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 47–66. Springer, Heidelberg (2012)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions (2004)
5. Cooper, K.D., Harvey, T.J., Kennedy, K.: A simple, fast dominance algorithm (2000), [www.cs.rice.edu/~keith/Embed/dom.pdf](http://www.cs.rice.edu/~keith/Embed/dom.pdf)
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 451–490 (1991)
7. Georgiadis, L., Tarjan, R.E.: Dominator tree verification and vertex-disjoint paths. In: SODA 2005, pp. 433–442 (2005)
8. Georgiadis, L., Werneck, R.F., Tarjan, R.E., August, D.I.: Finding Dominators in Practice. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 677–688. Springer, Heidelberg (2004)
9. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. J. ACM 23(1), 158–171 (1976)
10. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206 (1973)
11. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1, 121–141 (1979)
12. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
13. Ramalingam, G.: On loops, dominators, and dominance frontiers. ACM Trans. Program. Lang. Syst. 24(5), 455–490 (2002)
14. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL 2012 (2012)

# On the Correctness of an Optimising Assembler for the Intel MCS-51 Microprocessor<sup>\*</sup>

Dominic P. Mulligan and Claudio Sacerdoti Coen

Dipartimento di Scienze dell'Informazione,  
Università degli Studi di Bologna

**Abstract.** We present a proof of correctness in Matita for an optimising assembler for the MCS-51 microcontroller. The efficient expansion of pseudoinstructions, namely jumps, into machine instructions is complex. We isolate the decision making over how jumps should be expanded from the expansion process itself as much as possible using ‘policies’, making the proof of correctness for the assembler more straightforward.

Our proof strategy contains a tracking facility for ‘good addresses’ and only programs that use good addresses have their semantics preserved under assembly, as we observe that it is impossible for an assembler to preserve the semantics of every assembly program. Our strategy offers increased flexibility over the traditional approach to proving the correctness of assemblers, wherein addresses in assembly are kept opaque and immutable. In particular, we may experiment with allowing the benign manipulation of addresses.

**Keywords:** Verified software, CerCo (Certified Complexity), MCS-51 microcontroller, Matita proof assistant.

## 1 Introduction

We consider the formalisation of an assembler for the Intel MCS-51 8-bit microprocessor in the Matita proof assistant [1]. This formalisation forms a major component of the EU-funded CerCo (‘Certified Complexity’) project [3], concerning the construction and formalisation of a concrete complexity preserving compiler for a large subset of the C programming language.

The MCS-51 dates from the early 1980s and is commonly called the 8051/8052. Derivatives are still widely manufactured by a number of semiconductor foundries, with the processor being used especially in embedded systems.

The MCS-51 has a relative paucity of features compared to its more modern brethren, with the lack of any caching or pipelining features meaning that timing of execution is predictable, making the MCS-51 very attractive for CerCo’s ends. However, the MCS-51’s paucity of features—though an advantage in many respects—also quickly becomes a hindrance, as the MCS-51 features a relatively

---

<sup>\*</sup> The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

minuscule series of memory spaces by modern standards. As a result our C compiler, to be able to successfully compile realistic programs for embedded devices, ought to produce ‘tight’ machine code.

To do this, we must solve the ‘branch displacement’ problem—deciding how best to expand pseudojumps to labels in assembly language to machine code jumps. The branch displacement problem arises when pseudojumps can be expanded in different ways to real machine instructions, but the different expansions are not equivalent (e.g. differ in size or speed) and not always correct (e.g. correctness is only up to global constraints over the compiled code). For instance, some jump instructions (short jumps) are very small and fast, but they can only reach destinations within a certain distance from the current instruction. When the destinations are too far away, larger and slower long jumps must be used. The use of a long jump may augment the distance between another pseudojump and its target, forcing another long jump use, in a cascade. The job of the optimising compiler (assembler) is to individually expand every pseudo-instruction in such a way that all global constraints are satisfied and that the compiled program is minimal in size and faster in concrete time complexity. This problem is known to be computationally hard for most CISC architectures (see [4]).

To simplify the CerCo C compiler we have chosen to implement an optimising assembler whose input language the compiler will target. Labels, conditional jumps to labels, a program preamble containing global data and a MOV instruction for moving this global data into the MCS-51’s one 16-bit register all feature in our assembly language. We further simplify by ignoring linking, assuming that all our assembly programs are pre-linked.

Another complication we have addressed is that of the cost model. CerCo imposes a cost model on C programs or, more specifically, on simple blocks of instructions. This cost model is induced by the compilation process itself, and its non-compositional nature allows us to assign different costs to identical C statements depending on how they are compiled. In short, we aim to obtain a very precise costing for a program by embracing the compilation process, not ignoring it. At the assembler level, this is reflected by our need to induce a cost model on the assembly code as a function of the assembly program and the strategy used to solve the branch displacement problem. In particular, our optimising assembler should also return a map that assigns a cost (in clock cycles) to every instruction in the source program. We expect the induced cost to be preserved by the assembler: we will prove that the compiled code tightly simulates the source code by taking exactly the predicted amount of time.

Note that the temporal tightness of the simulation is a fundamental prerequisite of the correctness of the simulation because some functions of the MCS-51—timers and I/O—depend on the microprocessor’s clock. If the pseudo- and concrete clock differ the result of an I/O operation may not be preserved.

Branch displacement algorithms must have a deep knowledge of the way the rest of the assembler works in order to build globally correct solutions. Proving their correctness is quite a complex task (see, for instance, the companion paper [2]). Nevertheless, the correctness of the whole assembler only depends on

the correctness of the branch displacement algorithm. Therefore, in the rest of the paper, we presuppose the existence of a correct policy, to be computed by a branch displacement algorithm if it exists. A policy is the decision over how any particular jump should be expanded; it is correct when the global constraints are satisfied. The assembler fails to assemble an assembly program if and only if a correct policy does not exist. This is stated in an elegant way in the dependent type of the assembler: the assembly function is total over a program, a policy and the proof that the policy is correct for that program.

A final complication in the proof is due to the kind of semantics associated to pseudo-assembly programs. Should assembly programs be allowed to freely manipulate addresses? The traditional answer is ‘no’: values stored in memory or registers are either concrete data or symbolic addresses. The latter can only be manipulated in very restricted ways and programs that do not do so are not assigned a semantics and cannot be reasoned about. All programs that have a semantics have it preserved by the assembler. We take an alternative approach, allowing programs to freely manipulate addresses non-symbolically but only granting a preservation of semantics to those programs that act in ‘well-behaved’ ways. In principle, this should allow some reasoning on the actual semantics of malign programs. In practice, we note how our approach facilitates more code reuse between the semantics of assembly code and object code.

The formalisation of the assembler and its correctness proof are given in Sect. 2. Sect. 3 presents the conclusions and relations with previous work.

*Matita.* Matita is a proof assistant based on a variant of the Calculus of (Co)inductive Constructions [1]. It features dependent types that we exploit in the formalisation. The (simplified) syntax of the statements and definitions in the paper should be self-explanatory. Pairs are denoted with angular brackets,  $\langle -, - \rangle$ .

Matita features a liberal system of coercions. It is possible to define a uniform coercion  $\lambda x. \langle x, ? \rangle$  from every type  $T$  to the dependent product  $\Sigma x : T. P x$ . The coercion opens a proof obligation that asks the user to prove that  $P$  holds for  $x$ . When a coercion must be applied to a complex term (a  $\lambda$ -abstraction, a local definition, or a case analysis), the system automatically propagates the coercion to the sub-terms. For instance, to apply a coercion to force  $\lambda x. M : A \rightarrow B$  to have type  $\forall x : A. \Sigma y : B. P x y$ , the system looks for a coercion from  $M : B$  to  $\Sigma y : B. P x y$  in a context augmented with  $x : A$ . This is significant when the coercion opens a proof obligation, as the user will be presented with multiple, but simpler proof obligations in the correct context. In this way, Matita supports the ‘Russell’ proof methodology developed by Sozeau in [12], with an implementation that is lighter and more tightly integrated with the system than that of Coq.

## 2 Certification of an Optimising Assembler

Our aim here is to explain the main ideas and steps of the certified proof of correctness for an optimising assembler for the MCS-51.

In Subsect. 2.1 we sketch an operational semantics (a realistic and efficient emulator) for the MCS-51. We also introduce a syntax for decoded instructions that will be reused for the assembly language.

In Subsect. 2.2 we describe the assembly language and its operational semantics. The latter is parametric in the cost model that will be induced by the assembler, reusing the semantics of the machine code on all ‘real’ instructions.

Branch displacement policies are introduced in Subsect. 2.3 where we also describe the assembler as a function over policies as previously described.

To prove our assembler correct we show that the object code given in output, together with a cost model for the source program, simulates the source program executed using that cost model. The proof can be divided into two main lemmas. The first is correctness with respect to fetching, described in Subsect. 2.4. Roughly it states that a step of fetching at the assembly level, returning the decoded instruction  $I$ , is simulated by  $n$  steps of fetching at the object level that returns instructions  $J_1, \dots, J_n$ , where  $J_1, \dots, J_n$  is, amongst the possible expansions of  $I$ , the one picked by the policy. The second lemma states that  $J_1, \dots, J_n$  simulates  $I$  but only if  $I$  is well-behaved, i.e. manipulates addresses in ‘good’ ways. To keep track of well-behaved address manipulations we record where addresses are currently stored (in memory or an accumulator). We introduce a dynamic checking function that inspects this map to determine if the operation is well-behaved, with an affirmative answer being the pre-condition of the lemma. The second lemma is detailed in Subsect. 2.5 where we also establish correctness of our assembler as a composition of the two lemmas: programs that are well-behaved when executed under the cost model induced by the compiler are correctly simulated by the compiled code.

## 2.1 Machine Code and Its Semantics

We implemented a realistic and efficient emulator for the MCS-51 microprocessor. An MCS-51 program is just a sequence of bytes stored in the read-only code memory of the processor, represented as a compact trie of bytes addressed by the program counter. The **Status** of the emulator is a record that contains the microprocessor’s program counter, registers, stack pointer, clock, special function registers, data memory, and so on. The value of the code memory is a parameter of the record since it is not changed during execution.

The **Status** records is itself an instance of a more general datatype **PreStatus** that abstracts over the implementation of code memory in order to reuse the same datatype for the semantics of the assembly language in the next section.

The execution of a single instruction is performed by the `execute_1` function, parametric over the content `cm` of the code memory:

```
definition execute_1: ∀cm. Status cm → Status cm
```

The function `execute_1` closely matches the fetch-decode-execute cycle of the MCS-51 hardware, as described by a Siemen’s manufacturer’s data sheet [11]. Fetching and decoding are performed simultaneously: we first fetch, using the

program counter, from code memory the first byte of the instruction to be executed, decoding the resulting opcode, fetching more bytes as is necessary to decode the arguments. Decoded instructions are represented by the `instruction` data type which extends a data type of `preinstructions` that will be reused for the assembly language.

```

inductive preinstruction (A: Type[0]): Type[0] :=
| ADD: [[acc_a]] → [[registr; direct; indirect; data]] → preinstruction A
| DEC: [[acc_a; registr; direct; indirect]] → preinstruction A
| JB: [[bit_addr]] → A → preinstruction A
| ...
inductive instruction: Type[0] :=
| LCALL: [[addr16]] → instruction
| AJMP: [[addr11]] → instruction
| RealInstruction: preinstruction [[relative]] → instruction.
| ...

```

The MCS-51 has many operand modes, but an unorthogonal instruction set: every opcode is only enable for a finite subset of the possible operand modes. Here we exploit dependent types and an implicit coercion to synthesise the type of arguments of opcodes from a vector of names of operand modes. For example, `ACC` has two operands, the first one constrained to be the `A` accumulator, and the second one to be a disjoint union of register, direct, indirect and data operand modes.

The parameterised type `A` of `preinstruction` represents the addressing mode allowed for conditional jumps; in the `RealInstruction` constructor we constraint it to be a relative offset. A different instantiation (labels) will be used in the next section for assembly programs.

Once decoded, execution proceeds by a case analysis on the decoded instruction, following the operation of the hardware. For example, the `DEC` preinstruction (‘decrement’) is executed as follows:

```

| DEC addr ⇒
  let s := add_ticks1 s in
  let (result, flags) := sub_8_with_carry (get_arg_8 s true addr)
    (bitvector_of_nat 8 1) false in
    set_arg_8 s addr result

```

Here, `add_ticks1` models the incrementing of the internal clock of the micro-processor; it is a parameter of the semantics of `preinstructions` that is fixed in the semantics of `instructions` according to the manufacturer datasheet.

## 2.2 Assembly Code and Its Semantics

An assembly program is a list of potentially labelled pseudoinstructions, bundled with a preamble consisting of a list of symbolic names for locations in data memory (i.e. global variables). All preinstructions are pseudoinstructions, but



conditional jumps are now only allowed to use `Identifiers` (labels) as their target.

```

inductive pseudo_instruction: Type[0] :=
  | Instruction: preinstruction Identifier → pseudo_instruction
  ...
  | Jump: Identifier → pseudo_instruction
  | Call: Identifier → pseudo_instruction
  | Mov: [[dptr]] → Identifier → pseudo_instruction.

```

The pseudoinstructions `Jump`, `Call` and `Mov` are generalisations of machine code unconditional jumps, calls and move instructions respectively, all of whom act on labels, as opposed to concrete memory addresses. The object code calls and jumps that act on concrete memory addresses are ruled out of assembly programs not being included in the preinstructions (see previous Section).

Execution of pseudoinstructions is an endofunction on `PseudoStatus`. A `PseudoStatus` is an instance of `PreStatus` that differs from a `Status` only in the datatype used for code memory: a list of optionally labelled pseudoinstructions versus a trie of bytes. The `PreStatus` type is crucial for sharing the majority of the semantics of the two languages.

Emulation for pseudoinstructions is handled by `execute_1_pseudo_instruction`:

```

definition execute_1_pseudo_instruction:
  ∀cm. ∀costing:(∀ppc: Word. ppc < |snd cm| → nat × nat).
  ∀s:PseudoStatus cm. program_counter s < |snd cm| → PseudoStatus cm

```

The type of `execute_1_pseudo_instruction` is more involved than that of `execute_1`. The first difference is that execution is only defined when the program counter points to a valid instruction, i.e. it is smaller than the length `|snd cm|` of the program. The second difference is the abstraction over the cost model, abbreviated here as *costing*. The costing is a function that maps valid program counters to pairs of natural numbers representing the number of clock ticks used by the pseudoinstructions stored at those program counters. For conditional jumps the two numbers differ to represent different costs for the ‘true branch’ and the ‘false branch’. In the next section we will see how the optimising assembler induces the only costing (induced by the branch displacement policy deciding how to expand pseudojumps) that is preserved by compilation.

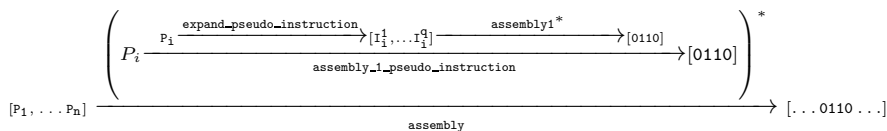
Execution proceeds by first fetching from pseudo-code memory using the program counter—treated as an index into the pseudoinstruction list. This index is always guaranteed to be within the bounds of the pseudoinstruction list due to the dependent type placed on the function. No decoding is required. We then proceed by case analysis over the pseudoinstruction, reusing the code for object code for all instructions present in the MCS-51’s instruction set. For all newly introduced pseudoinstructions, we simply translate labels to concrete addresses before behaving as a ‘real’ instruction.

We do not perform any kind of symbolic execution, wherein data is the disjoint union of bytes and addresses, with addresses kept opaque and immutable. Labels are immediately translated before execution to concrete addresses, and registers and memory locations only ever contain bytes, never labels. As a consequence, we allow the programmer to mangle, change and generally adjust addresses as they want, under the proviso that the translation process may not be able to preserve the semantics of programs that do this. This will be further discussed in Subsect. 2.5. The only limitation introduced by this approach is that the size of assembly programs is bounded by  $2^{16}$ .

### 2.3 The Assembler

The assembler takes in input an assembly program made of pseudoinstructions and a branch displacement policy for it. It returns both the object code (a list of bytes to be loaded in code memory for execution) and the costing for the source.

Conceptually the assembler works in two passes. The first pass expands every pseudoinstruction into a list of machine code instructions using the function `expand_pseudo_instruction`. The policy determines which expansion among the alternatives will be chosen for pseudo-jumps and pseudo-calls. Once the expansion is performed, the cost of the pseudoinstruction is defined as the cost of the expansion. The second pass encodes as a list of bytes the expanded instruction list by mapping the function `assembly1` across the list, and then flattening.



In order to understand the type for the policy, we briefly hint at the branch displacement problem for the MCS-51. A detailed description is found in [2]. The MCS-51 features three unconditional jump instructions: LJMP and SJMP—‘long jump’ and ‘short jump’ respectively—and an 11-bit oddity of the MCS-51, AJMP. Each of these three instructions expects arguments in different sizes and behaves in markedly different ways: SJMP may only perform a ‘local jump’ to an address closer then  $2^7$  bytes; LJMP may jump to any address in the MCS-51’s memory space and AJMP may jump to any address in the current memory page. Memory pages partition the code memory into  $2^8$  disjoint areas. The size of each opcode is different, with long jumps being larger than the other two. Because of the presence of AJMP, an optimal global solution may be locally unoptimal, employing a long jump where a shorter one could be used to force later jumps to stay inside single memory pages.

Similarly, a conditional pseudojump must be translated potentially into a configuration of machine code instructions, depending on the distance to the jump’s target. For example, to translate a jump to a label, a single conditional jump pseudoinstruction may be translated into a block of three real instructions as follows (here, JZ is ‘jump if accumulator is zero’):

	JZ <code>label</code>		JZ	size of SJMP instruction	
	...	translates to		SJMP	size of LJMP instruction
<code>label:</code>	MOV A B	$\implies$		LJMP	address of <code>label</code>
				...	
				MOV	A B

Naturally, if `label` is ‘close enough’, a conditional jump pseudoinstruction is mapped directly to a conditional jump machine instruction; the above translation only applies if `label` is not sufficiently local.

The cost returned by the assembler for a pseudoinstruction is set to be the cost of its expansion in clock cycles. For conditional jumps that are expanded as just shown, the costs of taking the true and false branches are different and both need to be returned.

The `expand_pseudo_instruction` function is driven by a policy in the choice of expansion of pseudoinstructions. The simplest idea is then to define policies as functions that maps jumps to their size. This simple idea, however, is impractical because short jumps require the offset of the target. For instance, suppose that at address `ppc` in the assembly program we found `Jmp l` such that `l` is associated to the pseudo-address `a` and the policy wants the `Jmp` to become a `SJMP  $\delta$` . To compute  $\delta$ , we need to know what the addresses `ppc+1` and `a` will become in the assembled program to compute their difference. The address `a` will be associated to is a function of the expansion of all the pseudoinstructions between `ppc` and `a`, which is still to be performed when expanding the instruction at `ppc`.

To solve the issue, we define the policy `policy` as a map from a valid pseudo-address to the corresponding address in the assembled program. Therefore,  $\delta$  in the example above can be computed simply as `policy(a) - policy(ppc + 1)`. Moreover, the `expand_pseudo_instruction` emits a `SJMP` only after verifying for each `Jmp` that  $\delta < 128$ . When this is not the case, the function emits an `AJMP` if possible, or an `LJMP` otherwise, therefore always picking the locally best solution. In order to accommodate those optimal solutions that require local sub-optimal choices, the policy may also return a Boolean used to force the translation of a `Jmp` into a `LJMP` even if  $\delta < 128$ . An essentially identical mechanism exists for call instructions and conditional jumps.

In order for the translation of a jump to be correct, the address associated to `a` by the policy and by the assembler must coincide. The latter is the sum of the size of all the expansions of the pseudo-instructions that precede the one at address `a`: the assembler just concatenates all expansions sequentially. To grant this property, we impose a correctness criterion over policies. A policy is correct when `policy(0) = 0` and for all valid pseudoaddresses `ppc`

$$\text{policy}(\text{ppc}+1) = \text{policy}(\text{ppc}) + \text{instruction\_size}(\text{ppc}) \leq 2^{16}$$

Here `instruction_size(ppc)` is the size in bytes of the expansion of the pseudoinstruction found at `ppc`, i.e. the length of `assembly_1_pseudo_instruction(ppc)`.

## 2.4 Correctness of the Assembler with Respect to Fetching

We now begin the proof of correctness of the assembler. Correctness consists of two properties: firstly that the assembly process never fails when fed a correct policy and secondly the object code returned simulates the source code when the latter is executed according to the cost model also returned by the assembler. This second property can be further decomposed into two main properties: correctness with respect to fetching and decoding and correctness with respect to execution.

Informally, correctness with respect to fetching is the following statement: when we fetch an assembly pseudoinstruction  $I$  at address  $\text{ppc}$ , then we can fetch the expanded pseudoinstruction(s)  $[J_1, \dots, J_n] = \text{fetch\_pseudo\_instruction } \dots I \text{ ppc}$  from policy  $\text{ppc}$  in the code memory obtained by loading the assembled object code. This section reviews the main steps to prove correctness with respect to fetching. Subsect. 2.5 deals with correctness with respect to execution: the instructions  $[J_1, \dots, J_n]$  simulate the pseudoinstruction  $I$ .

The (slightly simplified) Russell type for the `assembly` function is:

```

definition assembly:
  ∀program: pseudo_assembly_program. ∀policy.
    Σassembled: list Byte × (BitVectorTrie nat 16).
      |program| ≤ 216 → policy is correct for program →
        policy (|program|) = |fst assembled| ≤ 216 ∧
        ∀ppc: pseudo_program_counter. ppc < 216 →
          let pseudo_instr := fetch from program at ppc in
            let assembled_i := assemble pseudo_instr in
              |assembled_i| ≤ 216 ∧
              ∀n: nat. n < |assembled_i| → ∃k: nat.
                nth assembled_i n = nth assembled (policy ppc + k).

```

In plain words, the type of `assembly` states the following. Given a correct policy for the program to be assembled, the assembler never fails and returns some object code and a costing function. Under the condition that the policy is ‘correct’ for the program and the program is fully addressable by a 16-bit word, the object code is also fully addressable by a 16-bit word. Moreover, the result of assembling the pseudoinstruction obtained fetching from the assembly address  $\text{ppc}$  is a list of bytes found in the generated object code starting from the object code address  $\text{policy}(\text{ppc})$ .

Essentially the type above states that the `assembly` function correctly expands pseudoinstructions, and that the expanded instruction reside consecutively in memory. The fundamental hypothesis is correctness of the policy which allows us to prove the inductive step of the proof, which proceeds by induction over the assembly program. It is then straightforward to lift the property from lists of bytes (object code) to tries of bytes (i.e. code memories after loading). The `assembly_ok` lemma does the lifting.

We have established that every pseudoinstruction is compiled to a sequence of bytes that is found in memory at the expect place. This does not trivially imply that those bytes will be decoded in a correct way to recover the pseudoinstruction expansion. Indeed, we first need to prove a lemma that establishes that the `fetch` function is the left inverse of the `assembly1` function:

```
lemma fetch_assembly:
  ∀pc: Word.
  ∀i: instruction.
  ∀code_memory: BitVectorTrie Byte 16.
  ∀assembled: list Byte.
  assembled = assemble i →
  let len := |assembled| in
  let pc_plus_len := pc + len in
  encoding_check pc pc_plus_len assembled →
  let ⟨instr, pc', ticks⟩ := fetch pc in
  instr = i ∧ ticks = (ticks_of_instruction instr) ∧ pc' = pc_plus_len.
```

We read `fetch_assembly` as follows. Any time the encoding `assembled` of an instruction `i` is found in code memory starting at position `pc` (the hypothesis `encoding_check ...`), when we fetch at address `pc` retrieving the instruction `i`, the new program counter is `pc` plus the length of the encoding, and the cost of the fetched instruction is the one predicted for `i`. Or, in plainer words, assembling, storing and then immediately fetching gets you back to where you started.

Remembering that `assembly1_pseudo_instruction` is the composition of `assembly1` with `expand_pseudo_instruction`, we can lift the previous result from instructions (already expanded) to pseudoinstructions (to be expanded):

```
lemma fetch_assembly_pseudo:
  ∀program: pseudo_assembly_program.
  ∀policy, ppc, code_memory.
  let ⟨preamble, instr_list⟩ := program in
  let pi := π1 (fetch_pseudo_instruction instr_list ppc) in
  let pc := policy ppc in
  let instructions := expand_pseudo_instruction policy ppc pi in
  let ⟨l, a⟩ := assembly1_pseudoinstruction policy ppc pi in
  let pc_plus_len := pc + l in
  encoding_check code_memory pc pc_plus_len a →
  fetch_many code_memory pc_plus_len pc instructions.
```

Here, `l` is the number of machine code instructions the pseudoinstruction at hand has been expanded into. We assemble a single pseudoinstruction with `assembly1_pseudoinstruction`, which internally calls `expand_pseudo_instruction`. The function `fetch_many` fetches multiple machine code instructions from code memory and performs some routine checks.

Intuitively, Lemma `fetch_assembly_pseudo` says that expanding a pseudoinstruction into  $n$  instructions, encoding the instructions and immediately fetching  $n$  instructions back yield exactly the expansion.

Combining `assembly_ok` with the previous lemma and a proof of correctness of loading object code in memory, we finally get correctness of the assembler with respect to fetching:

```
lemma fetch_assembly_pseudo2:
  ∀program. |snd program| ≤ 216 →
  ∀policy. policy is correct for program →
  ∀ppc. ppc < |snd program| →
  let (assembled, costs') := π1 (assembly program policy) in
  let cmem := load_code_memory assembled in
  let (pi, newppc) := fetch_pseudo_instruction program ppc in
  let instructions := expand_pseudo_instruction policy ppc pi in
  fetch_many cmem (policy newppc) (policy ppc) instructions.
```

Here we use  $\pi_1$  to project the existential witness from the Russell-typed function `assembly`. We read `fetch_assembly_pseudo2` as follows. Suppose we are given an assembly program which can be addressed by a 16-bit word and a policy that is correct for this program. Suppose we are able to successfully assemble an assembly program using `assembly` and produce a code memory, `cmem`. Then, fetching a pseudoinstruction from the pseudo-code memory stored in the interval  $[ppc, newppc]$  corresponds to fetching a sequence of instructions from the real code memory, stored in the interval  $[policy(ppc), policy(ppc+1)]$ . The correspondence is precise: the fetched instructions are exactly those obtained expanding the pseudoinstruction according to policy.

In order to complete the proof of correctness of the assembler, we need to prove that each pseudoinstruction is simulated by the execution of its expansion (correctness with respect to execution). In general this is not the case when instructions freely manipulate program addresses. Characterising well-behaved programs and proving correctness with respect to expansion is discussed next.

## 2.5 Correctness for ‘Well-Behaved’ Assembly Programs

Most assemblers can map a single pseudoinstruction to zero or more machine instructions, whose size (in bytes) is not independent of the expansion. The assembly process therefore always produces a map (which for us is just the policy) that associates to each assembly address `a` a code memory address `policy(a)` where the instructions that correspond to the pseudoinstruction at `a` are located. Ordinarily, the map is not just a linear function, but depends on the local choices and global optimisations performed.

During execution of assembly code, addresses can be stored in memory locations or in the registers. Moreover, arithmetical operations can be applied to addresses, for example to compare them or to shift a function pointer in order to implement `C switch` statements. In order to show that the object code simulates the assembly code we must compute the processor status that corresponds to the assembly status. In particular, those `a` in memory that are used as data

should be preserved as `a`, but those used as addresses should be changed into `policy(a)`. Moreover, every arithmetic operation should commute with `policy` in order for the semantics to be preserved.

Following the previous observation, we can ask if it is possible at all for an assembler to preserve the semantics of an assembly program. The traditional approach to the verification of assemblers answers the question in the affirmative by restricting the semantics of assembly programs. In particular, the type of memory cells and registers is set to the disjoint union of data and symbolic addresses, and the semantics is always forced to consider all possible combinations of arguments (data vs. data, data vs. addresses, and so on), rejecting operations whose semantics cannot be preserved.

$$\text{Mem} : \text{Addr} \rightarrow \text{Bytes} + \text{Addr} \quad \llbracket - \rrbracket : \text{Instr} \rightarrow \text{Mem} \rightarrow \text{option Mem}$$

$$\llbracket \text{MUL } @A1 \ @A2 \rrbracket^M = \begin{cases} \text{Byte } b1, \text{ Byte } b2 & \rightarrow \text{Some}(M \text{ with accumulator } := b1 + b2) \\ -, \text{ Addr } a & \rightarrow \text{None} \\ \text{Addr } a, - & \rightarrow \text{None} \end{cases}$$

This approach has two main limitations. The first one is that it does not assign any semantics to interesting programs that could intentionally mangle addresses for malign (e.g. viruses) or benign (e.g. operating systems) purposes. The second is that it does not allow one to adequately share the semantics of assembly pseudoinstructions and object code instructions: only the `Byte-Byte` branch above can share the semantics with the object code `MUL`.

In this paper we have already taken a different approach from Sect. 2.2, where we have assigned a semantics to every assembly program by not distinguishing at all between data and symbolic addresses. Memory cells and registers always hold bytes, and symbolic labels are mapped to absolute addresses before execution. Consequently we do not expect that all assembly programs will have their semantics respected by object code. We call those programs that do *well-behaved*. Further, we can now reason over the semantics of programs that are not well-behaved, and that we can handle well-behavedness as an open predicate, recognising more and more good behaviours as required. Naturally, compilers that target our assembler will need to produce well-behaved programs, which is usually the case by construction.

The definition of well-behavedness we employ uses a map to keep track of the memory locations and registers that hold addresses during execution of an assembly program. The map acts as a sort of dynamic typing system sitting atop memory. This approach seems similar to one taken by Tuch *et al* [13] for reasoning about low-level C code.

The semantics of an assembly program is then augmented with a function that at each execution step updates the map, signalling an error when the program performs an ill-behaved operation. The actual computation is not performed by this mechanism, being already part of the assembly semantics.

$\text{AddrMap} : \text{Addr} \rightarrow \{\text{Data}, \text{Addr}\} \quad \llbracket - \rrbracket : \text{Instr} \rightarrow \text{AddrMap} \rightarrow \text{option AddrMap}$

$$\llbracket \text{MUL } @A1 @A2 \rrbracket^M = \begin{cases} \text{Data}, \text{Data} & \rightarrow \text{Some}(M \text{ with accumulator } := \text{Data}) \\ -, \text{Addr } a & \rightarrow \text{None} \\ \text{Addr } a, - & \rightarrow \text{None} \end{cases}$$

To prove semantic preservation we must associate an object code status to each assembly pseudostatus. This operation is driven by the current `AddrMap`: if at address `a` the assembly level memory holds `d`, then if `AddrMap(a) = Data` the object code memory will hold `d` (data is preserved), otherwise it will hold `policy(d)`. If all the operations accepted by the address update map are well-behaved, this is sufficient to show preservation of the semantics for those computation steps that are well-behaved, i.e. such that the map update does not fail.

We now apply the previous idea to the MCS-51, an 8-bit processor whose code memory is word addressed. All MCS-51 operations can therefore only manipulate and store one half of the address at a time (lower or higher bits). For instance, a memory cell could contain just the lower 8 bits of an address `a`. The corresponding cell at object code level must therefore hold the lower 8 bits of `policy(a)`, which can be computed only if we can also retrieve the higher 8 bits of `a`. We achieve this by storing the missing half of an address in the `AddrMap` — called `internal_pseudo_address_map` in the formalisation.

```
definition address_entry := upper_lower × Byte.
definition internal_pseudo_address_map :=
  (BitVectorTrie address_entry 7) × (BitVectorTrie address_entry 7)
  × (option address_entry).
```

Here, `upper_lower` is an inductive type with two constructors: `Upper` and `Lower`. The map consists of three components to track addresses in lower and upper internal ram and also in the accumulator `A`. If an assembly address `a` holds `h` and if the current `internal_pseudo_address_map` maps `a` to `(Upper, 1)`, then `h` is the upper part of the `h·1` address and `a` will hold the upper part of `policy(h·1)` in the object code status.

The relationship between assembly pseudostatus and object code status is computed by the following function which deterministically maps each pseudostatus into a corresponding status. It takes in input the policy and both the current pseudostatus and the current tracking map in order to identify those memory cells and registers that hold fragments of addresses to be mapped using `policy` as previously explained. It also calls the assembler to replace the code memory of the assembly status (i.e. the assembly program) with the object code produced by the assembler.

```
definition status_of_pseudo_status :
  internal_pseudo_address_map → ∀pap. ∀ps: PseudoStatus pap.
  ∀policy. Status (code_memory_of_pseudo_assembly_program pap policy)
```



The function that implements the tracking map update, previously denoted by  $\llbracket - \rrbracket$ , is called `next_internal_pseudo_address_map` in the formalisation. For the time being, we accept as good behaviours address copying amongst memory cells and the accumulator (MOV pseudoinstruction) and the use of the CJNE conditional jump that compares two addresses and jumps to a given label if the two labels are equal. Moreover, RET to return from a function call is well-behaved iff the lower and upper parts of the return address, fetched from the stack, are both marked as complementary parts of the same address (i.e. `h` is tracked as  $\langle \text{Upper}, 1 \rangle$  and `l` is tracked as  $\langle \text{Lower}, h \rangle$ ). These three operations are sufficient to implement the backend of the CerCo compiler. Other good behaviours could be recognised in the future, for instance in order to implement the C branch statement efficiently.

```

definition next_internal_pseudo_address_map: internal_pseudo_address_map →
  ∀cm. (Identifier → PseudoStatus cm → Word) → ∀s: PseudoStatus cm.
  program_counter s < 216 → option internal_pseudo_address_map

```

We now state the (simplified) statement of correctness of our compiler, whose proofs combines correctness with respect to fetching and correctness with respect to execution. It states that the well-behaved execution of a single assembly pseudoinstruction according to the cost model induced by compilation is correctly simulated by the execution of (possibly) many machine code instructions.

```

theorem main_thm:
  ∀M, M': internal_pseudo_address_map.
  ∀program: pseudo_assembly_program.
  ∀program_in_bounds: |program| ≤ 216.
  ∀policy. policy is correct for program.
  ∀ps: PseudoStatus program. ps < |program|.
  next_internal_pseudo_address_map M program ... = Some M' →
    ∃n. execute n (status_of_pseudo_status M ps policy) =
      status_of_pseudo_status M'
      (execute_1_pseudo_instruction program (ticks_of program policy) ps)
      policy.

```

The statement is standard for forward simulation, but restricted to `PseudoStatuses` `ps` whose tracking map is `M` and who are well-behaved according to `internal_pseudo_address_map` `M`. The `ticks_of program policy` function returns the costing computed by assembling the `program` using the given `policy`. An obvious corollary of `main_thm` is the correct simulation of  $n$  well-behaved steps by some number of steps  $m$ , where each step must be well-behaved with respect to the tracking map returned by the previous step.

### 3 Conclusions

We are proving the correctness of an assembler for MCS-51 assembly language. Our assembly language features labels, arbitrary conditional and unconditional

jumps to labels, global data and instructions for moving this data into the MCS-51's single 16-bit register. Expanding these pseudoinstructions into machine code instructions is not trivial, and the proof that the assembly process is 'correct', in that the semantics of a subset of assembly programs are not changed is complex.

The formalisation is a component of CerCo which aims to produce a verified concrete complexity preserving compiler for a large subset of the C language. The verified assembler, complete with the underlying formalisation of the semantics of MCS-51 machine code, will form the bedrock layer upon which the rest of CerCo will build its verified compiler platform.

We may compare our work to an 'industrial grade' assembler for the MCS-51: SDCC [10], the only open source C compiler that targets the MCS-51 instruction set. It appears that all pseudojumps in SDCC assembly are expanded to LJMP instructions, the worst possible jump expansion policy from an efficiency point of view. Note that this policy is the only possible policy *in theory* that makes every assembly program well-behaved, preserving its the semantics during the assembly process. This comes at the expense of assembler completeness as the generated program may be too large for code memory, there being a trade-off between the completeness of the assembler and the efficiency of the assembled program. The definition and proof of a terminating, correct jump expansion policy is described elsewhere [2].

Verified assemblers could also be applied to the verification of operating system kernels and other formalised compilers. For instance the verified seL4 kernel [5], CompCert [7] and CompCertTSO [14] all explicitly assume the existence of trustworthy assemblers. The fact that an optimising assembler cannot preserve the semantics of all assembly programs may have consequences for these projects.

Our formalisation exploits dependent types in different ways and for multiple purposes. The first purpose is to reduce potential errors in the formalisation of the microprocessor. Dependent types are used to constrain the size of bitvectors and tries that represent memory quantities and memory areas respectively. They are also used to simulate polymorphic variants in Matita, in order to provide precise typings to various functions expecting only a subset of all possible addressing modes that the MCS-51 offers. Polymorphic variants nicely capture the absolutely unorthogonal instruction set of the MCS-51 where every opcode must accept its own subset of the 11 addressing mode of the processor.

The second purpose is to single out sources of incompleteness. By abstracting our functions over the dependent type of correct policies, we were able to manifest the fact that the compiler never refuses to compile a program where a correct policy exists. This also allowed to simplify the initial proof by dropping lemmas establishing that one function fails if and only if some previous function does so.

Finally, dependent types, together with Matita's liberal system of coercions, allow us to simulate almost entirely in user space the proof methodology 'Russell' of Sozeau [12]. Not every proof has been carried out in this way: we only used this style to prove that a function satisfies a specification that only involves that function in a significant way. It would not be natural to see the proof that fetch and assembly commute as the specification of one of the two functions.

*Related work.* We are not the first to consider the correctness of an assembler for a non-trivial assembly language. The most impressive piece of work in this domain is Piton [8], a stack of verified components, written and verified in ACL2, ranging from a proprietary FM9001 microprocessor verified at the gate level, to assemblers and compilers for two high-level languages—Lisp and  $\mu$ Gypsy [9]. Klein and Nipkow also provide a compiler, virtual machine and operational semantics for the Jinja [6] language and prove that their compiler is semantics and type preserving.

Though other verified assemblers exist what sets our work apart from that above is our attempt to optimise the generated machine code. This complicates a formalisation as an attempt at the best possible selection of machine instructions must be made—especially important on devices with limited code memory. Care must be taken to ensure that the time properties of an assembly program are not modified by assembly lest we affect the semantics of any program employing the MCS-51's I/O facilities. This is only possible by inducing a cost model on the source code from the optimisation strategy and input program.

*Resources.* Our source files are available at <http://cerco.cs.unibo.it>. We assumed several properties of ‘library functions’, e.g. modular arithmetic and datastructure manipulation. We axiomatised various small functions needed to complete the main theorems, as well as some ‘routine’ proof obligations of the theorems themselves, in focusing on the main meat of the theorems. We believe that the proof strategy is sound and that all axioms can be closed, up to minor bugs that should have local fixes that do not affect the global proof strategy.

The complete development is spread across 29 files with around 20,000 lines of Matita source. Relevant files are: `AssemblyProof.ma`, `AssemblyProofSplit.ma` and `AssemblyProofSplitSplit.ma`, consisting of approximately 4500 lines of Matita source. Numerous other lines of proofs are spread all over the development because of dependent types and the Russell proof style, which does not allow one to separate the code from the proofs. The low ratio between source lines and the number of lines of proof is unusual, but justified by the fact that the pseudo-assembly and the assembly language share most constructs and large swathes of the semantics are shared.

## References

1. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. *Automated Reasoning* 39, 109–139 (2007)
2. Boender, J., Sacerdoti Coen, C.: On the correctness of a branch displacement algorithm (2012), <http://arxiv.org/abs/1209.5920>
3. The CerCo FET-Open project (2011), <http://cerco.cs.unibo.it/>
4. Branch displacement optimisation (2006), <http://groups.google.com/group/alt.lang.asm/msg/d31192d442accad3>
5. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating system kernel. In: *SOSP* (2009)

6. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems* 28(4), 619–695 (2006)
7. Leroy, X.: A formally verified compiler back-end. *Automated Reasoning* 43(4), 363–446 (2009)
8. Moore, J.S.: Piton: A mechanically verified assembly language. *Automated Reasoning Series*, vol. 3. Springer (1996)
9. Moore, J.S.: A grand challenge proposal for formal methods (2005)
10. Small device C compiler 3.0.0 (2011), <http://sdcc.sourceforge.net/>
11. Siemens Semiconductor Group 8051 derivative instruction set (2011), <http://www.win.tue.nl/~aeb/comp/8051/instruction-set.pdf>
12. Sozeau, M.: Subset Coercions in COQ. In: Altenkirch, T., McBride, C. (eds.) *TYPES 2006*. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007)
13. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: *POPL*, pp. 97–108 (2007)
14. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: *POPL*, pp. 43–54 (2011)

# An Executable Semantics for CompCert C<sup>\*</sup>

Brian Campbell

LFCS, University of Edinburgh  
Brian.Campbell@ed.ac.uk

**Abstract.** CompCert is a C compiler developed by Leroy et al, the majority of which is formalised and verified in the Coq proof assistant. The correctness theorem is defined in terms of a semantics for the ‘CompCert C’ language, but how can we gain faith in those semantics? We explore one approach: building an equivalent executable semantics that we can check test suites of code against.

Flaws in a compiler are often reflected in the output they produce: buggy compilers produce buggy code. Moreover, bugs can evade testing when they only manifest themselves on source code of a particular shape, when particular optimisations are used. This has made compilers an appealing target for mechanized verification, from early work such as Milner and Weyhrauch’s simple compiler in LCF [15] to modern work on practical compilers, such as the Verisoft C0 compiler [10] which is used as a key component of a larger verified software system.

The CompCert project [11] has become a nexus of activity in recent years, including the project members’ own efforts to refine the compiler and add certified optimisations [22,18,8], and external projects such as an extensible optimisation framework [20] and compiling concurrent programs [23]. The main result of the project is a C compiler which targets the assembly languages for a number of popular processor architectures. Most of the compiler is formalised in the Coq proof assistant and accompanied by correctness results, most notably proving that any behaviour given to the assembly output must be an acceptable behaviour for the C source program.

To state these correctness properties requires giving some semantics to the source and target languages. In CompCert these are given by inductively defined small-step relations. Errors and omissions in these semantics can weaken the overall theorems, potentially masking bugs in the compiler. In particular, if no semantics is given to a legitimate C program then any behaviour is acceptable for the generated assembly, and the compiler is free to generate bad code [1]. This

---

\* The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

<sup>1</sup> By design, the compiler is also free to fail with an error for *any* program, even if it is well-defined.

corresponds to the C standard’s notion of *undefined behaviour*, but this notion is left implicit in the semantics, so there is a danger of underdefining C.

Several methods could be used to gain faith in the semantics; in the conclusions of [11] Leroy suggests manual review, testing of executable versions and proving connections to alternative forms of semantics. Here we investigate testing an executable semantics which, by construction, is closely and formally related to CompCert’s input language. The testing is facilitated by Coq’s extraction mechanism, which produces OCaml code corresponding to the executable semantics that can be integrated with CompCert’s existing parser to produce a C interpreter.

Our motivation for this work comes principally from the CerCo project where we use executable semantics throughout [1], and in particular have an executable version of CompCert’s Clight intermediate language [5]. There we found an omission involving function pointers during testing. We wished to attempt to replicate this in CompCert C and search for any other evident problems. An executable semantics is far more practical for this task than manual animation of the inductive definitions<sup>2</sup>. Moreover, CompCert C is a more complex and interesting language than Clight, especially due to the presence of side-effects and non-determinism in the evaluation of expressions.

The main contributions of this paper are

1. demonstrating the successful retrofitting of a small-step executable semantics to an existing verified compiler with equivalence proofs against the original semantics,
2. showing that testing of the resulting interpreter does lead to the identification of bugs in both the semantics and the compiler,
3. showing that the executable semantics can illustrate the limitations of the semantics, and fill in a gap in the relationship between the original deterministic and non-deterministic versions of the semantics, and
4. demonstrate that a mixture of formal Coq and informal OCaml code can make testing more effective by working around known bugs with little effort.

The full development is available online as a modified version of CompCert<sup>3</sup>.

In Section 1 we will give an overview of the relevant parts of the CompCert compiler. In Section 2 we will discuss the construction of the executable semantics, its relationship to the inductively defined semantics from CompCert (both formalised in Coq) and the additional OCaml code used to support testing. This is followed by the results of testing the semantics in Section 3, including descriptions of the problems encountered. Finally, Section 4 discusses related work, including an interpreter Leroy added to newer versions of CompCert following this work.

---

<sup>2</sup> This is partly because manual animation requires providing witnesses for the results or careful management of existential metavariables, and partly because we can prove results once for the executable semantics rather than once per program. Automatic proof search has similar problems.

<sup>3</sup> <http://homepages.inf.ed.ac.uk/bcampbe2/compcert/>

# 1 Overview of the CompCert Compiler

CompCert compiles a simplified but substantial subset of the C programming language to one of three different architectures (as of version 1.8.2, which we refer to throughout unless stated otherwise). The main stages of the compiler, from an abstract syntax tree of the ‘CompCert C’ language to target specific pseudo-assembly code, are written in the Calculus of Inductive Constructions (CIC) — the dependently-typed language that underlies the Coq proof assistant [21]. The formal proofs of correctness refer to these definitions. Coq’s extraction facilities [13] produce OCaml code corresponding to them, which can be combined with a C parser and assembler text generation to complete the compiler. The resulting compilation chain is summarised in Figure 1.

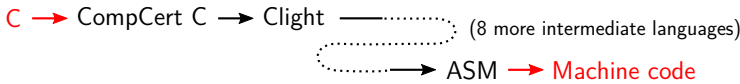


Fig. 1. Start and end of the CompCert compilation chain

Note that the unformalised C parser is not a trivial piece of code. Originally based on the CIL library for C parsing [17] but heavily adapted, it not only produces an abstract syntax tree, but also calculates type information and can perform several transformations to provide partial support for features that are not present in the formalised compiler. In particular, bitfields, structure passing and assignment, and **long double** and **long long** types are (optionally) handled here. There is a further transformation to simplify the annotated C abstract syntax tree into the CompCert C language.

Earlier versions of CompCert did not feature CompCert C at all, but the less complex Clight language. Clight lives on as the next intermediate language in the compiler. Hence several publications on CompCert refer to this language [3]. The major difference between the languages is that the syntax and semantics of expressions in Clight are much simpler because they are deterministic and side-effect free. Moreover, CompCert’s original semantics were in big-step form and lacked support for `goto` statements. Version 1.5 added these using a small-step semantics, and 1.8 added the CompCert C language with its C-like expressions. The latter effectively moved some of the work done by the OCaml parser into the formalised compiler.

CompCert C is also interesting because it has two semantics: one with non-deterministic behaviour for expressions which models the uncertainty about evaluation order in C, and a deterministic semantics which resolves this uncertainty. The former can be regarded as the intended input language of the compiler, and the latter as a more specific version that the compiler actually implements. In addition, these semantics have slightly different forms: in particular the non-deterministic version is always small-step, whereas the deterministic form makes

a single big-step for side-effect free subexpressions. The two are connected by a result showing that programs which are *safe* in the non-deterministic semantics admit some execution in the deterministic semantics, where by *safe* we mean that the program never gets stuck regardless of the order of evaluation used (for a fixed choice of I/O behaviour). This result essentially holds because the deterministic semantics is choosing one of the possible evaluation orders.

Note that throughout the definitions of all of these semantics we benefit from a specialisation of C to the targets that CompCert supports. Various *implementation-defined* parts of the language such as integer representation and sizes are fixed (in fact, the formalisation heavily uses the assumption of a 32-bit word size). Also, some *undefined behaviour* is given a meaning; notably mixtures of reads and writes to a variable in a single expression are given a non-deterministic behaviour rather than ruled out as per the C standard [4, §6.5.16].

## 2 Construction of the Executable Semantics

The executable semantics consists of several parts: existing executable definitions from CompCert, functions which closely correspond to the original relational semantics, a mechanism for resolving non-deterministic behaviour, and finally the proofs that steps of the relational semantics are equivalent to steps of the executable semantics. We do not consider whole program executions or interaction with the outside world (I/O) here, but only individual steps of the semantics. It is not difficult to include these (we have done this for Clight as part of the CerCo project [5]), but we do not expect that it would yield any worthwhile insights during testing to justify the effort involved arising from the extra non-determinism from I/O and the coinductive reasoning for non-terminating programs.

The existing definitions that we can reuse include the memory model that is used throughout the CompCert development [12]. This model features symbolic pointers with discrete abstract blocks of memory, and so is more suitable for our purposes than non-executable alternatives such as a concrete model with non-deterministic choice of fresh locations for allocations. The semantics of operations on values (for example,  $+$  and  $==$ ) are naturally defined as auxiliary functions in the relational semantics, which we reuse. Casts and the conversion of values to booleans are exceptions to this; they are defined inductively in the relational semantics and we treat them in the same way as the rest of the inductive definitions, below.

Local and global variable environments are necessarily executable in CompCert because they are used throughout the actual compiler code in addition to the semantics. We also make use of the compiler’s error monad to recover the partiality of the relational semantics (that is, CIC is a language of total functions, so we must model failure explicitly).

The environments are efficiently implemented because of their use in the compiler itself. However, the memory model was not intended to be efficient and builds large chains of closures to represent changes to memory. Fortunately, the performance was sufficient for our testing, so we did not attempt to optimise it.



## 2.1 Executing a Step of the Semantics

The relational semantics uses a series of inductive definitions to describe the steps of the abstract machine: one for reducing *lvalues*, expressions which represent a location that can be read from, written to, or extracted as a pointer; one for *rvalues* expressions that only yield a value, such as addition; and one for setting up function calls:

```

Inductive lred: expr -> mem -> expr -> mem -> Prop := ...
Inductive rred: expr -> mem -> expr -> mem -> Prop := ...
Inductive called: expr -> fundef -> list val -> type -> Prop := ...

```

The `estep` relation represents whole state transitions corresponding to the reductions on expressions given by the first three relations, and the `sstep` relation performs steps of statements and function entry and exit:

```

Inductive estep: state -> trace -> state -> Prop := ...
Inductive sstep: state -> trace -> state -> Prop := ...

```

The union of these two relations gives the overall `step` relation.

The executable semantics uses functions corresponding to each of these relations. The syntax directed nature of the inductive definitions makes this task straightforward. For example, consider the `rred` rules for conditional expressions:

```

| red_condition_true: forall v1 ty1 r1 r2 ty m,
  is_true v1 ty1 -> typeof r1 = ty ->
  rred (Econdition (Eval v1 ty1) r1 r2 ty) m
  (Eparen r1 ty) m
| red_condition_false: forall v1 ty1 r1 r2 ty m,
  is_false v1 ty1 -> typeof r2 = ty ->
  rred (Econdition (Eval v1 ty1) r1 r2 ty) m
  (Eparen r2 ty) m

```

These state that a true conditional reduces to its left subexpression and a false one to its right, subject to a typing constraint.

We rearrange the rules to fit in a tree of pattern matching, check type equalities where necessary, and introduce error messages where no reduction is possible:

```

Definition exec_rred (e:expr) (m:mem) : res (expr * mem) :=
match e with
...
| Econdition (Eval v1 ty1) r1 r2 ty =>
  do b <- exec_bool_val v1 ty1;
  let r := if b then r1 else r2 in
  match type_eq (typeof r) ty with
  | left _ => OK (Eparen r ty, m)
  | right _ => Error (msg "type mismatch in Econdition")
end

```

This transforms the value `v1` to a boolean, selects the appropriate subexpression and checks its type. The ‘do’ notation is a use of the error monad: the conversion to a boolean is not always defined and may return an error, which the

monad propagates to the caller of `exec_rred`. The auxiliary relations defining the relationship between values and booleans, `exec_bool_val`, and casting are also defined as functions.

Defining a function for the statement reduction, `sstep`, is similar. One part of function entry handling deserves particular attention. CompCert has a notion of *external functions*, which are a set of primitive CompCert C functions that are handled outside of the program to provide system calls, dynamic memory allocation, volatile memory accesses and annotations. Inductive predicates describe what changes to the program’s state are allowed. We have only implemented the dynamic memory allocation (by constructing a function similar to the predicates for `malloc` and `free`), and provide a predicate on states to identify uses of the other calls so that we may give a partial completeness result in the next section.

The expression relation `estep` is more difficult. This is the point in the semantics at which non-deterministic behaviour appears — any subexpression in a suitable *context* can be reduced. The contexts are defined by an inductive description of which subexpressions can be evaluated in the current state (for example, only the guard of a conditional expression can be evaluated before the conditional expression itself, whereas either subexpression of  $e_1 + e_2$  can be reduced). We could execute the non-deterministic semantics by collecting all of the possible reductions in a set (in the form of the possible successor states), but we are primarily interested in testing the semantics on well behaved C programs, and an existing theorem in CompCert shows that any program with a *safe* non-deterministic behaviour also has a behaviour in the deterministic semantics.

Hence we are satisfied with following a single path of execution, and so parametrise the executable semantics by a *strategy* function which determines the evaluation order by picking a particular subexpression, and also returns its context. These contexts should be valid with respect to the inductive definition mentioned above. Moreover, if we choose a function matching the deterministic semantics then we know that any well-defined C program which goes wrong is demonstrating a bug (in the form of missing behaviour) which affects *both* sets of semantics. Regardless of the choice of strategy, we can use failed executions to pinpoint rules in the semantics which concern us. We will discuss the implemented strategies and their implications in the following section.

Finally, to match the non-deterministic semantics of `estep` precisely we must also check that there are no *stuck* subexpressions. This detects subexpressions which cannot be reduced because they are erroneous, rather than because they are already values. Normally the lack of any reducible subexpression would indicate an error, but the extra check is required because an accompanying non-terminating subexpression will mask it. The CompCert documentation gives the example `f() + (10 / x)` where `x` is initially 0 and `f` is non-terminating. Without the check, a program containing this expression would appear to be well-defined because `f` can always be reduced, whereas we should make the program undefined because of the potential division by zero.

Informally, the condition for `estep` states that *any subexpression in an evaluation context is either a value, or has a further subexpression that is reducible.*

This form suggests an inefficient executable approach: for every context perform a search for the reducible subexpression. Fortunately it can be implemented by a relatively direct recursive function because we can reuse reducible subexpressions that we have already found in a larger context. Thus we only have to check that an expression can be reduced (using the reduction functions discussed above) when all of the subexpressions have already been reduced to values.

The deterministic semantics does not enforce this condition because it commits to one particular evaluation, regardless of whether other evaluations may fail. This means that the relationship between the non-deterministic and deterministic semantics is not a straightforward inclusion: the deterministic semantics accepts some executions where the stuckness check fails, but rejects the executions allowed by the non-deterministic semantics under a different expression reduction order. Thus we make the executable version of the check optional so that we may implement either semantics.

## 2.2 Equivalence to the Non-deterministic Semantics

We show the equivalence of the executable semantics to the original relational semantics in two parts, summarising the proofs from the formal Coq development. First, we show that any successful execution of a step corresponds to some derivation in the original semantics, regardless of the strategy we choose.

**Theorem 1 (Soundness).** *Given a strategy which picks valid contexts, the execution of any step  $s_1$  to  $s_2$  with trace  $t$  implies that there exists a derivation of step  $s_1$   $t$   $s_2$ .*

The proof consists of building similar lemmas for each relation in the semantics, using case analysis on the states, statements and expressions involved, reduction of the executable hypothesis and reasoning on equalities until the premises of the constructor are realised. The check for *stuck* subexpressions is the most involved task: we must prove a more general result differentiating between reducible and fully reduced expressions, and show that reducible terms found in subexpressions can be lifted. If no such term exists, we show that attempting to reduce the whole term is sufficient.

The second part is to show that any step of the original semantics is successfully executed by the executable semantics.

**Theorem 2 (Completeness — non-deterministic).** *For any derivation of step  $s_1$   $t$   $s_2$  which is not about to call an unsupported external function there exists a strategy  $p$  such that  $\text{exec\_step } p \ s_1 = \text{OK } (t, s_2)$ .*

As in the soundness proof, a lemma is shown for each relation. The general form of the proofs is to perform inversion on the hypothesis for the derivation and use rewriting and the prior lemmas to reduce the goal. Again, the stuckness check is more difficult: it follows by induction on the expression, and showing that if a subexpression gets stuck then the parent expression gets stuck too.

Note that each step might require a different strategy because we have restricted our attention to strategies that can be expressed as functions on the abstract machine state. This rules out strategies where the execution order depends on (for example) previous states or randomness, but these do not occur in the CompCert compiler.

### 2.3 Strategies and the Deterministic Semantics

Two evaluation strategies were implemented for the executable semantics. The first was a simple leftmost-innermost strategy that picks the first non-value subexpression that we are allowed to reduce. It was chosen because it is simple to implement, sufficient for testing (any bugs were likely to be independent of evaluation order, and our experience with the second strategy supported this), and could be used as the basis for the second strategy.

However, to provide the greatest benefit from the executable semantics we wanted to get executions which precisely match the deterministic semantics that the compiler actually implements. That is, we wish to have an interpreter that predicts exactly the behaviour of the compiled program. This is not entirely straightforward because the original deterministic semantics are formalised in a slightly different way to non-deterministic executions: expressions with no side effects are dealt with by a big-step relation, and effectful expressions use a small-step relation as before. Here ‘effectful’ includes both changes to the state and to control flow such as the conditional operator. This prioritisation of the effectful subexpressions is also what causes the difference in evaluation order. For example,  $x+(x=1)$  when  $x \neq 1$  gives a different answer with the second strategy because the assignment is evaluated first.

Despite the difference in construction, we can still encode the resulting evaluation order as a strategy:

1. find the leftmost-innermost *effectful* subexpression (or take the whole expression if it is effect-free), then
2. (a) pick the leftmost-innermost *effect-free* non-value subexpression of that if one is present, otherwise
  - (b) reduce the whole subexpression.

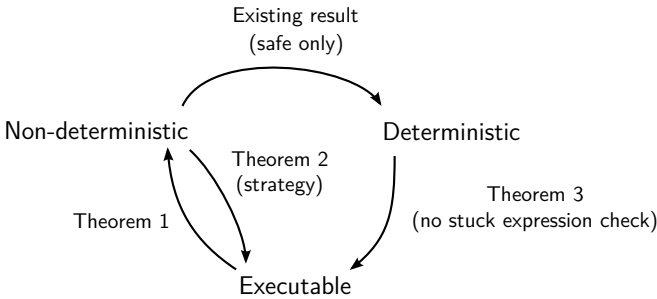
Intuitively, this works because the effect-free subexpressions can be reduced in any order that is allowed without changing the result.

To show formally that this matches the relational semantics we first show that the big-steps of effect-free subexpressions can be decomposed into a series of small-steps using the relations from the non-deterministic semantics. We can then show that iterating the executable semantics will perform the same steps, reaching the same end state as the big-step. Then using some lemmas to show that the `leftcontexts` (the deterministic version of expression contexts) used in the semantics are found by our leftmost-innermost effectful subexpression strategy, we can show that any step of the deterministic semantics is performed by some number of steps of the executable semantics:

**Theorem 3 (Completeness — deterministic).** *For any derivation of  $\text{step } s_1 \ t \ s_2$  in the deterministic semantics which is not about to call an unsupported external function, repeated use of the `exec_step` function starting with  $s_1$  using the above strategy yields the state  $s_2$ . Moreover, the concatenation of the resulting traces is  $t$ .*

Again, a precise version of the above argument is formalised in Coq.

Together with the connections between the non-deterministic semantics and the executable semantics, and the result from CompCert linking the two relational semantics we get the relationships depicted in Figure 2. We can see that the new result completes a loop — confirming the intuition that the deterministic semantics is the same as the non-deterministic semantics with a particular choice of evaluation order, but without the ‘stuckness-check’.



**Fig. 2.** Relationships between the different CompCert C semantics

We also see directly from the executable semantics that the evaluation order is the *only* possible source of non-determinism besides I/O, because each step of the semantics can be expressed as a function rather than a relation.

## 2.4 Informal OCaml Code

To produce the actual interpreter we add some OCaml code which uses the existing CompCert parser to produce CompCert C from C source code, and then iterates the step function extracted from Coq until the program successfully completes, or fails with an error.

Ideally we would fix each bug that we encounter when testing this interpreter, but this can involve repairing the specifications, the compiler, and the proofs; all of which CompCert’s developers are better placed to deal with. Nonetheless, we wish to continue testing in spite of any bugs found. To achieve this we detect troublesome states in the OCaml main loop and override the normal behaviour of the executable semantics without changing the formal development. To assist in the implementation of these workarounds we use a higher order function to

apply a local change to an expression to any applicable subexpression appearing in an evaluation context.

We also take the opportunity to add some support functions that are outside the scope of the semantics, but which are informally supported by the compiler. They are implemented by detecting states which call these functions and running an OCaml version instead. We provide `exit` and `abort` (trivial functions to halt execution), `memcpy` and `memcmp` (impossible to implement in CompCert C because the memory model does not provide byte-by-byte representations of pointers<sup>4</sup>) and a limited version of `printf`. CompCert’s semantics also assume that the entire program is present in a single file, so we provide some support for merging definitions from multiple files to simulate separate compilation.

All of these can be added without requiring any extra proof or changing the compiler or the semantics. They can also be turned off to be sure that problems we encounter come from the formalised semantics and not the workarounds.

Finally, we note that the execution of the semantics can be examined in the debugger packaged with OCaml when absolute certainty about the cause of a problem is required. Indeed, during the testing below the debugger was used to retrieve extra information about the program state that was not included in the interpreter’s error message.

### 3 Testing

After some basic testing to ensure that the interpreter was functioning correctly (in particular, testing the informal driver code), we proceeded with the example that illustrated an omission in CerCo’s Clight semantics, then attempted more demanding tests.

#### 3.1 Function Pointers

The simple program

```
int zero(void) { return 0; }

int main(void) {
  int (*f)(void) = zero;
  return f();
}
```

returns zero by a call to the `zero` function via a function pointer. It failed during testing of CerCo’s executable semantics due to a bad type check, and CompCert C was suspected to suffer from the same problem. Indeed, execution fails at the call with the same type error: the variable `f`’s type is a pointer to a function and the semantics only accepts a function type.

---

<sup>4</sup> Implementation of `memcpy` and `memcmp` was assisted by the fact that Coq’s extraction process exposes the internal representation of the memory model, rather than forcing us to implement them through the model’s interface.

There are two noteworthy aspects to this bug. First, the compiler itself features the correct type check and works perfectly. The error only affects the specification of the compiler; this is extra behaviour that has not been proved correct. The semantics can be easily modified to use the same check, and the only other change required is to *remove* some steps from the proof scripts. Second, the bug only became relevant from version 1.7 of CompCert — before which the parser always added an explicit dereference. This illustrates a general issue: the intended semantics of CompCert C is unclear because it is difficult to understand what we should assume about the output of the complex parser.

### 3.2 Csmith

Csmith is a tool for the random generation of test cases for C compilers that uses a mixture of static and dynamic checks to ensure that the generated code has well-defined behaviour [24]. Failing test cases are identified by the compiler failing with a crash or error, the generated code crashing, or by comparing the output of the test case with that obtained using different compilers. Each generated test case outputs a checksum to summarise its runtime behaviour for comparison.

Csmith is particularly interesting because it has already been used to test the CompCert compiler as a whole where it detected several bugs in the informal part of the compiler and a mismatch between the assembler output and the assembler’s behaviour. These problems have since been corrected, partly by the introduction of CompCert C as the input language. Thus it gives us the opportunity to compare the semantics against the actual compiler. However, it is not an ideal tool for testing semantics because it focuses on detecting ‘middle-end’ bugs, and thus only generates a limited range of front-end language features.

Given the previous testing of the compiler, it was unsurprising that we found no problems when executing the randomly generated code with the interpreter and comparing the results against the compiler<sup>5</sup>. However, we did encounter a failure with the *non-random* support code which implements the dynamic checks that prevent undefined behaviour in arithmetic operations. The failing code can be seen in Figure 3.

The failure is caused by the reduction rules for conditional expressions:

```
| red_condition_true: forall v1 ty1 r1 r2 ty m,
  is_true v1 ty1 -> typeof r1 = ty ->
  rred (Econdition (Eval v1 ty1) r1 r2 ty) m
  (Eparen r1 ty) m
| red_condition_false: forall v1 ty1 r1 r2 ty m,
  is_false v1 ty1 -> typeof r2 = ty ->
  rred (Econdition (Eval v1 ty1) r1 r2 ty) m
  (Eparen r2 ty) m
```

Each rule requires the type of the chosen subexpression to equal the type of the entire expression. However, here one branch is an 8-bit integer, and the other a 32-bit integer so one of the rules cannot be applied. The C standard requires

<sup>5</sup> Using Csmith version 2.0.

```

int8_t lshift_func_int8_t_s(int8_t left, int right)
{
  return
    ((left < 0) ||
     (((int)right) < 0) ||
     (((int)right) >= 32) ||
     (left > (INT8_MAX >> ((int)right)))) ?

  left :

  (left << ((int)right));
}

```

**Fig. 3.** Excerpt of the safe arithmetic code from Csmith (edited for readability)

that ‘the usual arithmetic conversions’ should be applied to coerce the result to a common type [4, §6.5.15], so the 8-bit integer should be promoted to a 32-bit one.

As with the function pointers, this test works with the compiler because the two types have a common representation — all integers are handled as 32-bit words in CompCert. It is only when they are loaded and stored to memory that the integer size is taken into account. Nonetheless, this failure provides us with enough information to construct an example on which the compiler fails because the representations differ:

```

double f(int x, int a, double b) {
  return x ? a : b;
}

```

Fortunately, the compiler performs some type reconstruction for the RTL intermediate language which makes it produce an error rather than bad code. Curiously, this means that the overall correctness theorem still holds if the semantics are corrected because the compiler is always allowed to fail, even on reasonable programs. However, the intermediate results about the front-end become false because the bug is present much earlier in the compiler than the type reconstruction.

Alleviating this bug was essential for proper testing with Csmith, but a full solution requires fixing the semantics, compiler and proofs and is beyond the scope of this work. By implementing an alternative reduction rule in OCaml as a workaround we were able to continue testing without difficulty.

### 3.3 gcc-torture

The GCC compiler contains an extensive test suite for several of its supported languages [7, §6.4]. The gcc-torture suite contains an executable subset of the tests in C. Many of the test cases use GCC-specific features, but we are able



to reuse a subset selected by another executable C semantics project, `kcc` [6]. Unlike the Csmith generated tests, `gcc-torture` contains many specialised tests for corner-cases of the C standard. In addition to finding errors, these also serve to highlight deliberate limitations in CompCert’s semantics.

A small test harness ran each test case with the executable semantics. The failing cases were manually classified and in a few cases a workaround or fix to the parser was added to prevent known bugs hiding further issues.

The tests revealed that the semantics were missing some minor casting rules and that zero initialisation of file scope variables was not performed when initialising the memory model. Both were handled correctly by the compiler, in the latter case by the informal code that produces the assembler output.

Several issues with the OCaml parser appeared: initialisation of an array by a shorter string did not add null-padding; arrays were not permitted to the left of a `->` dereference; incomplete array types were not fully supported; and the reported line numbers in error messages could be wrong. Ironically, the latter was caused by side-effects and OCaml’s own non-deterministic evaluation order.

Deliberate limitations of CompCert were strongly apparent in the result too. Unsupported and partially supported constructs such as `long long` integer types and bitfields caused many test case failures. A more interesting case is that the comparison of pointers is undefined when one points just beyond the end of the object (which is explicitly permitted by the C standard), or when a function pointer is used. However, we were unable to find any mention of this limitation in the documentation, and without testing would never have known about it.

## 4 Related Work

The CompCert developers had already proposed implementing an executable semantics for the first version of Clight, but in a big-step style with a bound on the depth of the evaluation to ensure termination [3, §5.4]. However, it would be difficult to implement the workarounds described in Section 2.4 with a big-step semantics.

Reaction to this work from the CompCert developers has been positive: bugs identified before the next release were fixed (in the case of the missing casts, independently before they were identified by testing), and Leroy has subsequently added a similar small-step interpreter. The new interpreter differs from the present work by focusing solely on the non-deterministic semantics and computing all of the successor states at once. This greatly simplifies the checking and proof of non-stuckness described in Section 2.1, and provides an option to explore the entire space of evaluation orders (which is surprisingly tractable). However, there is no easy way to mimic just the deterministic evaluation order, and no tricks in the driver code to make testing easier. Together with the bug fixes, this makes directly comparing test results with the current work infeasible, although some brief testing with the `gcc-torture` suite behaved as expected. The latest version of CompCert, 1.11, also features a more efficient memory model, improving the interpreter’s performance.

Ševčík et al. [23] have created CompCertTSO, a derivative of an earlier version of CompCert which supports concurrent shared-memory programs with a *Total Store Ordering* (TSO) memory model. The source language for the formalised part of the compiler is ClightTSO and is given an executable semantics in addition to the usual relational semantics, which ‘revealed a number of subtle errors.’ The main issues found in the present work did not arise because it was based on a version of CompCert that predated their introduction, and the pointer comparisons were not a problem because they were specified differently in CompCertTSO.

Ellison and Roşu [6] have developed a semantics for C based on rewriting logic that aims to be as close to the standard as possible, including many parts of the language that are not currently supported by CompCert. An interpreter called `kcc` is derived from the semantics and has been tested against the `gcc-torture` test suite that we reused. They go further and perform coverage analysis to gauge the proportion of the semantics exercised by the tests. These semantics are not used for compiler verification, but are intended for studying (and ultimately verifying) the behaviour of C programs.

The Piton compiler [16] was formalised in the Boyer-Moore prover, which is an example of an environment where an executable semantics is the natural choice. The Piton language is a very low-level systems language defined for the project, so they do not benefit from preexisting test suites. However, the work is particularly interesting due to the connected hardware formalisation of the target. A later use of executable semantics in ACL2 demonstrated the usefulness of executable semantics in hardware verification because running an existing test suite for AMD’s RTL language against the semantics was crucial for convincing managers that the model was relevant [19, §3].

Lochbihler and Bulwahn [14] applied and extended Isabelle’s code extraction and predicate compiler [2] to animate the semantics of a multithreaded Java formalisation in Isabelle/HOL, JinjaThreads. This is a very appealing approach because the predicate compiler deals with most of the burden of writing an executable version of an inductively defined semantics, although JinjaThreads still required a substantial amount of work to deal with some difficult definitions. Their description of testing focuses on performance rather than correctness, but executable versions of the code have been tested on an ongoing basis since the earlier Jinja formalisation that it is based on [9].

## 5 Conclusions

We have shown that executable semantics can be useful enough for the necessary task of validating semantics to justify retrofitting them to existing verified compilers, and in the case of CompCert found a real compiler bug in the formalised front-end, alongside numerous minor issues. It also illustrates that testing of the semantics can be more sensitive than testing the compiler: our original failing case for the conditional expressions bug would (and did) pass compiler testing,

but using that failure in the semantics we were able to derive a test case that also failed in the compiler.

In addition to the testing, the executable semantics were also useful for demonstrating the limitations of the semantics, both known and unknown. Moreover, we were able to take the opportunity to prove that an intuitive relationship between the deterministic and non-deterministic semantics of CompCert C holds.

## References

1. Amadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D., Pollack, R., Régis-Gianas, Y., Coen, C.S., Stark, I.: Certified complexity. *Procedia Computer Science* 7, 175–177 (2011)
2. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning Inductive into Equational Specifications. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 131–146. Springer, Heidelberg (2009), [http://dx.doi.org/10.1007/978-3-642-03359-9\\_11](http://dx.doi.org/10.1007/978-3-642-03359-9_11)
3. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 263–288 (2009), <http://dx.doi.org/10.1007/s10817-009-9148-3>
4. Programming languages — C. International standard ISO/IEC 9899:1999, ISO (1999)
5. Campbell, B., Pollack, R.: Executable formal semantics of C. Tech. Rep. EDI-INF-RR-1412, School of Informatics, University of Edinburgh (2010)
6. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*, pp. 533–544. ACM, New York (2012), <http://doi.acm.org/10.1145/2103656.2103719>
7. Free Software Foundation: GNU Compiler Collection (GCC) Internals, version 4.4.3 (2008)
8. Jourdan, J.-H., Pottier, F., Leroy, X.: Validating *LR*(1) Parsers. In: Seidl, H. (ed.) *ESOP 2012*. LNCS, vol. 7211, pp. 397–416. Springer, Heidelberg (2012), [http://dx.doi.org/10.1007/978-3-642-28869-2\\_20](http://dx.doi.org/10.1007/978-3-642-28869-2_20)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.* 28(4), 619–695 (2006), <http://doi.acm.org/10.1145/1146809.1146811>
10. Leinenbach, D., Petrova, E.: Pervasive compiler verification from verified programs to verified systems. *Electronic Notes in Theoretical Computer Science* 217, 23–40 (2008), <http://www.sciencedirect.com/science/article/pii/S1571066108003836>
11. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* 52, 107–115 (2009), <http://doi.acm.org/10.1145/1538788.1538814>
12. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41(1), 1–31 (2008)
13. Letouzey, P.: A New Extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) *TYPES 2002*. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
14. Lochbihler, A., Bulwahn, L.: Animating the Formalised Semantics of a Java-Like Language. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) *ITP 2011*. LNCS, vol. 6898, pp. 216–232. Springer, Heidelberg (2011), [http://dx.doi.org/10.1007/978-3-642-22863-6\\_17](http://dx.doi.org/10.1007/978-3-642-22863-6_17)

15. Milner, R., Weyhrauch, R.: Proving compiler correctness in a mechanized logic. *Machine Intelligence* 7, 51–70 (1972)
16. Moore, J.S.: A mechanically verified language implementation. *Journal of Automated Reasoning* 5, 461–492 (1989), <http://dx.doi.org/10.1007/BF00243133>
17. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: *CC 2002*. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
18. Rideau, S., Leroy, X.: Validating Register Allocation and Spilling. In: Gupta, R. (ed.) *CC 2010*. LNCS, vol. 6011, pp. 224–243. Springer, Heidelberg (2010), [http://dx.doi.org/10.1007/978-3-642-11970-5\\_13](http://dx.doi.org/10.1007/978-3-642-11970-5_13)
19. Moore, J.S.: Symbolic Simulation: An ACL2 Approach. In: Gopalakrishnan, G.C., Windley, P. (eds.) *FMCAD 1998*. LNCS, vol. 1522, pp. 334–350. Springer, Heidelberg (1998), [http://dx.doi.org/10.1007/3-540-49519-3\\_22](http://dx.doi.org/10.1007/3-540-49519-3_22)
20. Tatlock, Z., Lerner, S.: Bringing extensibility to verified compilers. In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, pp. 111–121. ACM, New York (2010), <http://doi.acm.org/10.1145/1806596.1806611>
21. Team, T.C.D.: *The Coq Proof Assistant: Reference Manual, Version 8.3*. INRIA (2010), <http://coq.inria.fr/distrib/8.3pl2/refman/>
22. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pp. 17–27. ACM, New York (2008)
23. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pp. 43–54. ACM, New York (2011), <http://doi.acm.org/10.1145/1926385.1926393>
24. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pp. 283–294. ACM, New York (2011), <http://doi.acm.org/10.1145/1993498.1993532>

# Producing Certified Functional Code from Inductive Specifications

Pierre-Nicolas Tollitte<sup>1</sup>, David Delahaye<sup>2</sup>, and Catherine Dubois<sup>3</sup>

<sup>1</sup> CEDRIC/ENSIIE, Évry, France  
tollitte@ensiie.fr

<sup>2</sup> CEDRIC/CNAM, Paris, France  
David.Delahaye@cnam.fr

<sup>3</sup> CEDRIC/ENSIIE/INRIA, Paris, France  
dubois@ensiie.fr

**Abstract.** Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). Both styles have pros and cons. Relational style may be preferred because it allows the user to describe only what is true, discard momentarily the termination question, and stick to a rule-based description. However, a relational specification is usually not executable. This paper proposes to turn an inductive specification into a functional one, in the logical setting itself, more precisely `Coq` in this work. We define for a certain class of inductive specifications a way to extract functions from them and automatically produce the proof of soundness of the extracted function w.r.t. its inductive specification. In addition, using user-defined modes which label inputs and outputs, we are able to extract several computational contents from a single inductive type.

**Keywords:** Executable Specifications, Inductive Relations, Functional Code Generation, Soundness Proof Generation, `Coq`.

## 1 Introduction

Proof assistants based on type theory allow the user to adopt either a functional style, or a relational style (e.g., by using inductive types). The choice between the two styles may be guided by different requirements, but it is also a matter of taste. Using inductive types or relational style may be preferred because it allows the user to specify only what is true, discard termination issues, and stick to usual rule-based presentations. A typical example, which illustrates these points, concerns the definition of an operational semantics including a while loop. While it is rather tricky to define an interpretation function [4], it is easier and more natural to define its operational semantics by means of an inductive relation. The former approach has to deal with general recursion, partiality, and termination. The latter approach provides the user with a straightforward implementation, where the inference system is formalized as an inductive type whose constructors are a direct rewording of the inference rules. However, in some systems like `Coq`

or Isabelle, these definitions are not directly executable. Simulating the execution of a program therefore requires proving a judgment using the constructors of the inductive type with more or less automation, and this kind of process does not scale up to complex specifications. Another argument in favor of relational style is that an inductive specification may describe several computational behaviors according to the arguments of the inductive relation which have been selected as inputs. For example, from the predicate *add* where *add n m p* specifies that *p* is the sum of *n* and *m*, it is possible to extract not only a function realizing the addition, but also a function realizing the subtraction.

A feature offered by some systems such as Coq, Isabelle, or HOL, consists of the possibility to extract code from functional specifications. In Isabelle [3,2] and Coq (see the previous work of some of the authors [6], as well as the plugin distributed with Coq), it is even possible to do so from inductive specifications. However, in the Coq framework, an ML function extracted from an inductive type, even if it terminates, cannot be used in the Coq environment itself.

In this paper, we propose an approach which turns an inductive specification into a functional one within the logical setting itself, i.e. Coq [10] in particular. For a class of inductive specifications, we define a way to extract functions from these specifications and produce the proof of soundness of any extracted function w.r.t. its inductive specification. This allows us to not only use the generated functions within the proof assistant, but also reason over them. This approach is fully automatic if the extracted function follows a structural recursion scheme. Otherwise, if the recursion is general, the user must provide termination information, such as a measure or a well-founded order. Our approach is limited to inductive specifications from which we can extract structurally recursive functions. Our contribution consists in automating a common but tedious practice, as illustrated by this quotation by Blazy and Leroy in [5]: “*The recommended approach to execute a Coq specification by inductive predicates, therefore, is to define a reference interpreter as a Coq function, prove its equivalence with the inductive specification, and evaluate applications of the function.*”.

To produce the Coq functional code, we follow the translation scheme given in [6] for extracting ML code from inductive specifications. Compared to [6], we are able to deal with a larger family of inductive specifications, involving in particular some specific cases of non-deterministic inductive relations, and as said previously, we are also able to deal with proofs of soundness.

The closest approach to the work described in this paper concerns the compilation of inductive relations in Isabelle/HOL into executable programs [3,2]. In this approach, the authors rely on a mode consistency analysis, in the same way as in [6] and the work presented here. The notion of mode comes from logic programming and helps us perform various analyses and optimizations. In [2], the inductive definition is translated into a set of equations equivalent to the initial moded definition, and then exported to a functional programming language. The equivalence is proved by means of a sound and complete procedure. The main difference between this approach and our work is that the authors are able to compile non-deterministic specifications, while we reject some of them. As a

consequence, this approach relies on an infrastructure of sequences in order to have all the possible results even if the specification is deterministic, resulting in less readable programs in some cases. In another context, inductive specifications encoded in Twelf [9] can be executed using a higher-order logic programming language, but it does not export any code within or outside of the logic.

The paper is structured as follows: in Section 2, we illustrate our approach on a basic example; we then introduce, in Section 3, our notion of inductive specification, and present our code generation algorithm; next, in Section 4, we describe the generation of proofs of soundness for the extracted functions w.r.t. their corresponding inductive specifications; finally, in Section 5, we provide information regarding the implementation which has been realized in the framework of Coq.

## 2 An Example

In this section, we present how our functional extraction should work on an example of inductive specification within the Coq framework [10]. This functional extraction is performed in several steps. First, the user annotates his/her inductive relation with a mode specifying which arguments are inputs, the others being considered as outputs. A mode consistency analysis is then performed to determine if the extraction is possible w.r.t. the provided mode. If the previous analysis is successful, the inductive relation is translated into a functional program. Finally, if the previous translation is successful, a proof of soundness is produced, ensuring that the generated function verifies the initial inductive relation. Compared to [6], the translation into a functional program is performed within the logical framework (i.e. Coq), which requires the extracted function to terminate and allows us to generate a proof of soundness.

As an example of extraction, let us consider the specification consisting in searching for a value in a binary search tree, which can be formalized in Coq as follows (let us note that we introduce two inductive relations, one for comparing two values of the tree, and another one for searching a value in the tree):

```
Inductive bst : Set :=
  | Empty : bst
  | Node : bst → nat → bst → bst.
```

```
Inductive comp_nat : Set := | Inf | Sup | Eq.
```

```
Inductive path : Set :=
  | Not_found | End_path
  | Left : path → path | Right : path → path.
```

```
Inductive compare : nat → nat → comp_nat → Prop :=
  | Compare_eq : compare 0 0 Eq
  | Compare_inf : forall n, compare 0 (S n) Inf
  | Compare_sup : forall n, compare (S n) 0 Sup
  | Compare_rec : forall n m c, compare n m c →
    compare (S n) (S m) c.
```

```

Inductive search : bst → nat → path → Prop :=
  | Search_empty : forall n, search Empty n Not_found
  | Search_found : forall n m t1 t2, compare n m Eq →
    search (Node t1 m t2) n End_path
  | Search_inf : forall n m t1 t2 b, search t1 n b →
    compare n m Inf → search (Node t1 m t2) n (Left b)
  | Search_sup : forall n m t1 t2 b, search t2 n b →
    compare n m Sup → search (Node t1 m t2) n (Right b).
    
```

Using the mode  $\{1,2\}$  both for the *compare* and *search* relations (which means that we use the two first arguments of *compare* and *search* as inputs), the following function can be automatically extracted from the relation *search*:

```

Fixpoint search12 (p1 : bst) (p2 : nat) : path :=
  match p1 with
  | Empty ⇒ Not_found
  | Node t1 m t2 ⇒
    match compare12 p2 m with
    | Inf ⇒ let b := search12 t1 p2 in Left b
    | Sup ⇒ let b := search12 t2 p2 in Right b
    | Eq ⇒ End_path
  end
end.
    
```

where *compare12* is the function extracted from the relation *compare*.

It should be noted that using the mode  $\{1,2\}$ , the relation *search* appears as non-deterministic in the sense that several constructors overlap (in this case, *Search\_found*, *Search\_inf*, and *Search\_sup*). This requires a specific analysis of the premises of the corresponding constructors to realize that the relation is actually deterministic using the result of a given call to distinguish them (here, the result of the application of *compare12*).

Once the previous function has been generated, it is possible to produce a proof of soundness for this function, i.e. to prove that it verifies the relation from which it has been extracted. To do so, the idea is to use the functional induction scheme of the extracted function generated by Coq, which is the following (due to space restrictions, we only describe the cases of *Search\_empty* and *Search\_inf*):

```

Lemma search12_ind :
  forall P : bst → nat → path → Prop,
    (forall (p1 : bst) (p2 : nat), p1 = Empty →
      P Empty p2 Not_found) →
    (forall (p1 : bst) (p2 : nat) (t1 : bst) (m : nat)
      (t2 : bst), p1 = Node t1 m t2 →
      compare12 p2 m = Inf → P t1 p2 (search12 t1 p2) →
      let p := search12 t1 p2 in
      P (Node t1 m t2) p2 (Left p)) → ...
  forall (p1 : bst) (p2 : nat), P p1 p2 (search12 p1 p2).
    
```



Using this induction scheme, it is possible to automatically complete the proof of soundness for the function previously extracted as follows (we still focus on the cases corresponding to the constructors *Search\_empty* and *Search\_inf*):

```

Lemma search12_sound :
  forall (p1 : bst) (p2 : nat) (p : path),
    search12 p1 p2 = p → search p1 p2 p.
Proof.
  intros until 0; intro H; subst p; apply search12_ind.
  (* Search_empty *)
  intros until 0; intro H; apply Search_empty.
  (* Search_inf *)
  intros until 0; intros H1 H2 H3; simpl.
  apply Search_inf;
  [assumption | apply compare12_sound; assumption].
  ... (* Search_sup and Search_found *)
Save.

```

where *compare12\_sound* is the soundness lemma for the *compare12* function.

### 3 Code Generation

Our code generation algorithm consists in producing a functional program from an inductive relation and an extraction mode. In the following, we will borrow some definitions and notations from [6], and in particular, an inductive relation will be called logical inductive type. If the extraction is performed from a logical inductive type  $d$ , the definition of  $d$  may use other logical inductive types named  $d_i$ . In this case, extraction modes must be provided for all these types. We will not deal with mutually recursive definitions, and we will assume that each dependency w.r.t.  $d_i$  has already been extracted with its extraction mode.

#### 3.1 Logical Inductive Types

The Coq proof assistant relies on the Calculus of Inductive Constructions (CIC for short) type theory, for which a description can be found in the Coq documentation [10]. This theory is actually too extensive for the purpose of this paper, and we will use a subset of CIC to describe logical inductive types. The subset of CIC that we will consider is very similar to the one used in [6], and we will only add some restrictions on the form of the terms. An inductive definition is noted  $\text{Ind}(d : \tau, \Gamma)$ , where  $d$  is the name of the inductive definition,  $\tau$  its type, and  $\Gamma$  the context representing the constructors (their names together with their respective types). In this notation, two restrictions have been made: we do not deal with parameters (i.e. the additional arguments which are shared by the type  $\tau$  of the inductive definition and the types of constructors defined in  $\Gamma$ ) and mutual inductive definitions. In addition, dependent types, higher order and propositional arguments are not allowed in the type of an inductive definition; more precisely, this means that  $\tau$  has the following form

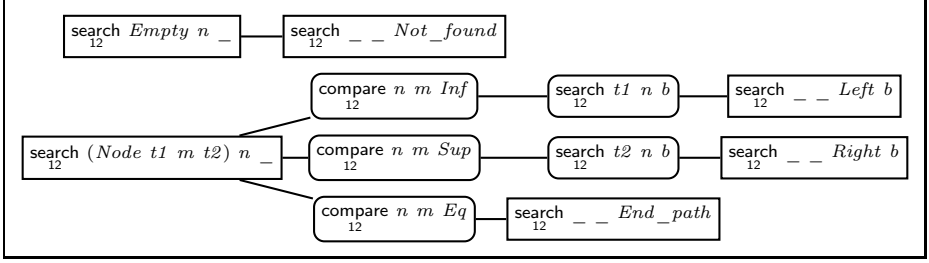
$\tau_1 \rightarrow \dots \tau_i \rightarrow \dots \tau_n \rightarrow \text{Prop}$  where  $\tau_i$ , with  $i = 1 \dots n$ , is of type **Set** or **Type**, and does not contain any product or dependent inductive type. Moreover, we suppose that the types of constructors are in prenex form, with no dependency between the bounded variables and no higher order; thus, the type of a constructor is  $\forall x_1 : X_1, \dots, x_n : X_n. T_1 \rightarrow \dots T_j \rightarrow \dots T_m \rightarrow d \ t_1 \dots t_p$  where  $x_i \notin X_l$ , with  $l > i$ ,  $X_i$  is of type **Set** or **Type**,  $T_j$  is of type **Prop** and does not contain any product or dependent inductive type, and  $t_k$ , with  $k = 1 \dots p$ , are terms. In the following, the terms  $T_j$  are called the premises of the constructor, whereas the term  $d \ t_1 \dots t_p$  is called the conclusion of the constructor. We impose the additional constraint that  $T_j$  is a fully applied logical inductive type, i.e.  $T_j$  is of the form  $d_j \ t_{j1} \dots t_{jp_j}$ , where  $d_j$  is a logical inductive type (possibly different from  $d$ ),  $t_{jk}$ , with  $k = 1 \dots p_j$ , are terms, and  $p_j$  is the arity of  $d_j$ . Additionally, we put some restrictions on the form of a term  $t_i$ , which is either a variable or a fully applied constructor  $c_i \ t_{i1} \dots t_{ip_i}$ , where  $p_i$  is the arity of  $c_i$ . An inductive type verifying the conditions above is called a logical inductive type. We aim to propose an extraction method for this kind of inductive types.

In the general case, we aim to extract only deterministic specifications. We actually distinguish two kinds of determinism. The basic notion of determinism is when for a given extraction mode, the inputs of the conclusions of constructors are pairwise non-unifiable. However, there also exists another kind of determinism, where the logical inductive type seems non-deterministic but actually remains deterministic, i.e. where there are overlapping conclusions of constructors, but where a function can still be extracted (see the example of Section 2). In contrary to 6, we propose to also deal with this other kind of determinism in some specific cases, where using a premise, we can distinguish between the constructors whose conclusions overlap. This requires the use of a specific representation of logical inductive types, which is introduced in the next subsection.

In the following, we will refer to the constructors using their names. In order to denote the constructor named  $C$ , we will use  $\Gamma(C)$ . We will also add the notation  $P(C)$  to denote the set of premises of a constructor named  $C$ , and the notation  $P(C, i)$  to denote the  $i^{\text{th}}$  premise of the constructor  $C$ .

### 3.2 Intermediate Representation for Merging Constructors

As said previously, the work developed in this paper also proposes to relax some restrictions imposed in 6. One of them is that constructors do not overlap, which means that their respective conclusions are pairwise non-unifiable. This restriction is too strong as it prevents from handling quite common specifications like the example of Section 2. However, when some conclusions overlap, we can still generate some code in some cases. The first pattern matching of the generated function is usually used to distinguish between constructors. To extract specifications with overlapping conclusions, the idea is to merge them and use premises to distinguish between constructors. In the example of Section 2, we have to merge the conclusions of the *Search\_inf*, *Search\_sup*, and *Search\_found* constructors. These three conclusions will be compiled as the same pattern in the extracted function. In some cases, it may be also necessary to merge premises.



**Fig. 1.** Rel-Tree Representation for the Binary Search Tree Example

**Relation-Tree Definition.** In order to represent the merging of constructors, we introduce a new intermediate data structure to represent logical inductive types where constructors can be merged. This new representation is based on trees, a data structure which eases both the verification of some properties over the specification and the code generation. Logical inductive types are actually represented as a forest called a relation-tree, which is defined as follows:

**Definition 1 (Relation-Tree).** *Given a logical inductive type  $d$ , it can be represented by the following relation-tree (or rel-tree for short):*

$$\text{Rel-Tree}(\{(d\ t_{i1}\ \dots\ t_{ip}, \text{Nodes}_1), \dots, (d\ t_{k1}\ \dots\ t_{kp}, \text{Nodes}_k)\})$$

where  $\text{Nodes}_i$  is either  $\{(T_{i1}, \text{Nodes}_{i1}), \dots, (T_{ik}, \text{Nodes}_{ik})\}$  or  $d\ t_{i1}\ \dots\ t_{ip}$ .

It should be noted that in this definition,  $k$  is always smaller than or equal to the number of constructors in the logical inductive type  $d$ , because there is at most one node by constructor, and less if there are merged constructors.

Considering the example of the binary search tree of Section 2, it is possible to represent the inductive relation *search* by the rel-tree of Figure 1. In this representation, we use several conventions. The leaves are represented at the right-hand side. The nodes annotated by a conclusion of the specification are represented by the boxes with sharp corners, whereas the nodes related to premises are in the boxes with rounded corners. In these nodes, some arguments may be hidden with underscores when they are not relevant; for instance, we hide the output in the root nodes and the inputs in the leaf nodes, because they are not involved when extracting the code. In addition, in each node, the extraction mode is mentioned under the considered relation name.

**Rel-Tree Properties.** The main task of the code generation is to build a rel-tree verifying the three properties described below (the code generation itself is in turn quite straightforward). In order to describe these three properties, we need a function to get a path from a rel-tree. In the following, the `treepaths` function will return the set of paths that can be built from a rel-tree. For example, if `bst_tree` denotes the rel-tree of Figure 1, we have:

$$\begin{aligned} \text{treepaths}(\text{bst\_tree}) = \\ \{ [ \text{search } \text{Empty } n \_ ; \text{search } \_ \_ \text{Not\_found} ], \\ [ \text{search } (\text{Node } t1 \ m \ t2) \ n \_ ; \text{compare } n \ m \ \text{Inf}; \text{search } t1 \ n \ b; \\ \text{search } \_ \_ \text{Left } b ], \dots \} \end{aligned}$$

We also need functions to compute input and output terms according to a mode, where a mode is a set of indexes which correspond to the arguments of the logical inductive type used as inputs:

**Definition 2 (Functions for Inputs/Outputs).** *Given a logical inductive type  $d$ , some terms  $t_1 \dots t_{p_d}$ , and a mode  $m$ , we define the following functions:*

$$\begin{aligned} \text{in}(d \ t_1 \ \dots \ t_{p_d}, m) &\triangleq (t_{i_1}, \dots, t_{i_m}), \text{ where } m \text{ is } \{i_1, \dots, i_m\} \\ \text{invars}(d \ t_1 \ \dots \ t_{p_d}, m) &\triangleq \text{variables}(\text{in}(d \ t_1 \ \dots \ t_{p_d}, m)) \\ \text{out}(d \ t_1 \ \dots \ t_{p_d}, m) &\triangleq \begin{cases} \text{if } \exists j \in 1 \dots p_d, j \notin \{i_1, \dots, i_m\} \text{ then } t_j \\ \text{else } \text{true} \end{cases} \\ \text{where } m \text{ is } &\{i_1, \dots, i_m\} \\ \text{outvars}(d \ t_1 \ \dots \ t_{p_d}, m) &\triangleq \text{variables}(\text{out}(d \ t_1 \ \dots \ t_{p_d}, m)) \end{aligned}$$

where  $\text{variables}(t)$  returns the set of variables occurring in the term  $t$ .

We also define the global environment  $\mathcal{M}$ , which contains the extraction modes for all the logical inductive types used in the logical inductive type being extracted. We can get the extraction mode for the logical inductive type  $d$  using the notation  $\mathcal{M}(d)$ . This notation is extended for the premises as follows: if  $T_i$  is of the form  $d_i \ t_{i1} \ \dots \ t_{ip_i}$ , then  $\mathcal{M}(T_i) = \mathcal{M}(d_i)$ . In the following,  $d$  and  $m$  will refer to the logical inductive type being extracted and its extraction mode.

The first property describes the relationship between the logical inductive type and its rel-tree. We must ensure that we find all constructors with their conclusions and their premises in the rel-tree (up to renaming).

*Property 1 (Specification Compliance).* The rel-tree  $r$  is said to comply with its logical inductive type  $d$  iff it verifies the following property:

$$\begin{aligned} \text{SC}(r) &\triangleq \forall C \in \Gamma, \exists! p \in \text{treepaths}(r), \text{SC}'(p, C) \wedge \\ &\forall p \in \text{treepaths}(r), \exists! C \in \Gamma, \text{SC}'(p, C) \end{aligned}$$

where  $\text{SC}'$  is the compliance of a path  $[T_0; T_1; \dots; T_n; T_{n+1}]$  for a given constructor named  $C$  defined as follows:

$$\begin{aligned} \text{SC}'([T_0; T_1; \dots; T_n; T_{n+1}], C) &\triangleq \\ \exists \sigma, \exists \Pi, n = \text{card}(P(C)) \wedge \forall i \in 1 \dots n, T_j = \sigma(P(C, \Pi i)) \wedge \\ \text{out}(T_{n+1}, m) = \text{out}(\sigma(\text{concl}(\Gamma(C))), m) \wedge \\ \text{in}(T_0, m) = \sigma(\text{in}(\text{concl}(\Gamma(C))), m) \end{aligned}$$

where  $\sigma$  is a variable renaming,  $\Pi$  a permutation of  $1 \dots n$ , and  $\text{concl}$  a function which returns the conclusion term of a constructor.

The second property is similar to the mode consistency analysis of [6], but is performed on the rel-tree instead of the logical inductive type. It verifies that variables are not used before they are defined in the generated function.

*Property 2 (Mode Consistency Analysis).* Given a mode  $m$  and a rel-tree  $r$  of the form  $\text{Rel-Tree}(\{(d\ t_{11} \dots t_{1p}, \text{Nodes}_1); \dots; (d\ t_{n1} \dots t_{np}, \text{Nodes}_n)\})$ ,  $m$  is said to be consistent w.r.t.  $r$  iff the following property is verified:

$$\text{MCA}(\{(d\ t_{11} \dots t_{1p}, \text{Nodes}_1); \dots; (d\ t_{n1} \dots t_{np}, \text{Nodes}_n)\}) \triangleq \forall i, i \in 1 \dots n, \text{MCA}'(\text{Nodes}_i, \text{invars}(d\ t_{i1} \dots t_{ip}))$$

where  $\text{MCA}'$  is defined as follows:

$$\text{MCA}'(\text{nodes}, S) \triangleq \begin{cases} \text{if } \text{nodes} \text{ is } d\ t_1 \dots t_p \text{ then} \\ \quad \text{outvars}(d\ t_1 \dots t_p, m) \subseteq S \\ \text{if } \text{nodes} \text{ is } \{(T_1, \text{Nodes}_1); \dots; (T_n, \text{Nodes}_n)\} \text{ then} \\ \quad \forall i, i \in 1 \dots n, \text{invars}(T_i, \mathcal{M}(T_i)) \subseteq S \wedge \\ \quad \text{MCA}'(\text{Nodes}_i, S \cup \text{outvars}(T_i, \mathcal{M}(T_i))) \end{cases}$$

The third property ensures that we build valid pattern matchings from the rel-tree, i.e. with exclusive clauses (involving non-overlapping patterns). The patterns of the same pattern matching will be the outputs of the premise nodes which are the children of the same parent node.

*Property 3 (Non-Overlapping Patterns).* Given a mode  $m$  and a rel-tree  $r$  of the form  $\text{Rel-Tree}(\{(d\ t_{11} \dots t_{1p}, \text{Nodes}_1); \dots; (d\ t_{n1} \dots t_{np}, \text{Nodes}_n)\})$ , the function extracted from  $r$  in mode  $m$  will only involve non-overlapping patterns iff the following property is verified:

$$\text{NO}(\{(d\ t_{11} \dots t_{1p}, \text{Nodes}_1); \dots; (d\ t_{n1} \dots t_{np}, \text{Nodes}_n)\}) \triangleq \forall i, i \in 1 \dots n, \text{NO}'(\text{Nodes}_i) \wedge \forall j, j \in 1 \dots n, j \neq i \Rightarrow \text{in}(d\ t_{i1} \dots t_{ip}, m) \text{ and } \text{in}(d\ t_{j1} \dots t_{jp}, m) \text{ are not unifiable}$$

where the  $\text{NO}'$  property is defined as follows:

$$\text{NO}'(\text{nodes}) \triangleq \begin{cases} \text{if } \text{nodes} \text{ is } d\ t_1 \dots t_p \text{ then } \text{true} \\ \text{if } \text{nodes} \text{ is } [(T_1, \text{Nodes}_1); \dots; (T_n, \text{Nodes}_n)] \text{ then} \\ \quad \forall i, i \in 1 \dots n, \text{NO}'(\text{Nodes}_i) \wedge \forall j, j \in 1 \dots n, j \neq i \Rightarrow \\ \quad \text{in}(T_i, \mathcal{M}(T_i)) = \text{in}(T_j, \mathcal{M}(T_j)) \wedge \\ \quad \text{out}(T_i, \mathcal{M}(T_i)) \text{ and } \text{out}(T_j, \mathcal{M}(T_j)) \text{ are not unifiable} \end{cases}$$

Due to space restrictions, we do not present the rel-tree generation algorithm in details here, and we only provide a short description of this algorithm. The complexity of this algorithm mainly comes from the possible permutations of premises. The basic idea is to generate all the possible rel-trees from the specification and find if there is any rel-tree verifying the three properties introduced

above. However, in order to generate fewer rel-trees (because there are many permutations of premises), we therefore add some heuristics using the three properties described above. From the first property, we can deduce a general form of rel-trees. Each rel-tree must contain one path for each constructor of the specification. Each path must begin and end with the conclusion. Except these two nodes, there is one node for each premise. The second property can be verified independently for each path. Only the third property needs all the constructors to be present in the rel-tree to be verified, but it can be verified after each insertion. As a result, we can insert the constructors one by one, and verify the three properties at each step.

### 3.3 Partial Mode Extraction of Complete Specifications

As said in the introduction, we only consider structurally recursive functions in this paper. As a consequence, the extracted functions are generated as regular fixpoints of CIC, which consists of our target language. We actually use the following subset of CIC (we use the notations of the Coq documentation [10]):

$$\begin{aligned}
 t ::= & \text{fix } f (x_1 : \tau_1) \dots (x_n : \tau_n) : \tau := t \\
 & | f \ t_1 \dots t_{p_f} \mid c \ t_1 \dots t_{p_c} \mid \text{let } x := t_1 \text{ in } t_2 \mid x \\
 & | (\text{match } t_m \text{ with} \\
 & \quad | c_1 \ x_{11} \dots x_{1p_1} \Rightarrow f_1 \mid \dots \mid c_n \ x_{n1} \dots x_{np_n} \Rightarrow f_n)
 \end{aligned}$$

In this language, there is no complex pattern matching. We can only match one term of type  $\tau$  and there is one pattern per constructor of the type  $\tau$ . However, to simplify the presentation of the code generation, we use more complex pattern matching expressions (with nested patterns) as follows:

$$\begin{aligned}
 & \text{match } (t_{m1}, \dots, t_{mn}) \text{ with} \\
 & | p_{11}, \dots, p_{1n} \Rightarrow t_1 \mid \dots \mid p_{k1}, \dots, p_{kn} \Rightarrow t_k
 \end{aligned}$$

where “ $p ::= c \ p_1 \dots p_{p_c} \mid x \mid \_$ ”.

These more complex pattern matching expressions are then compiled into simpler pattern matching expressions, which conform to the initial language of the CIC subset considered for the extraction. This compilation is performed using a specialized version of the algorithm described in [7].

In the following, we describe the code generation for partial modes and complete specifications. A mode is partial iff there is one output (otherwise, when there is no output, the mode is full). A specification is complete for a given mode iff its extraction produces a complete function, in which all the pattern matchings are exhaustive. The extraction for full modes and incomplete specifications will be explained later, as evolutions of the algorithm described below.

The code generation of a rel-tree  $r$  built from the inductive definition  $d$ , of the form  $\text{Rel-Tree}(\{(d \ t_{11} \dots t_{1p}, \text{Nodes}_1), \dots, (d \ t_{k1} \dots t_{kp}, \text{Nodes}_k)\})$ , and extracted with the mode  $m = \{i_1, \dots, i_m\}$ , is denoted by  $\llbracket r \rrbracket_{\mathcal{M}}$  and performed in the following way:

$$\begin{aligned} & \llbracket \{(d\ t_{11} \dots t_{1p}, \text{Nodes}_1), \dots, (d\ t_{k1} \dots t_{kp}, \text{Nodes}_k)\} \rrbracket_{\mathcal{M}} \triangleq \\ & \text{fix } f_d(x_1 : \tau_{i_1}) \dots (x_m : \tau_{i_m}) : \tau_o := \\ & \quad \text{match } (x_1, \dots, x_m) \text{ with} \\ & \quad | \text{in}(d\ t_{11} \dots t_{1p}) \Rightarrow \llbracket \text{Nodes}_1 \rrbracket_{\mathcal{M}} \\ & \quad | \dots \\ & \quad | \text{in}(d\ t_{k1} \dots t_{kp}) \Rightarrow \llbracket \text{Nodes}_k \rrbracket_{\mathcal{M}} \end{aligned}$$

This function generates the outermost pattern matching of the extracted function, and the generated code for each node is produced as follows:

$$\llbracket \text{Nodes} \rrbracket_{\mathcal{M}} \triangleq \left\{ \begin{array}{l} \text{if } \text{Nodes} \text{ is } d\ t_1 \dots t_p \text{ then } \text{out}(d\ t_1 \dots t_p, m) \\ \text{if } \text{Nodes} \text{ is } \{(T_{i1}, \text{Nodes}_{i1}), \dots, (T_{ik}, \text{Nodes}_{ik})\} \text{ then} \\ \quad \text{match } \llbracket T_{i1} \rrbracket_{\mathcal{M}} \text{ with} \\ \quad | \text{out}(T_{i1}, \mathcal{M}(T_{i1})) \Rightarrow \llbracket \text{Nodes}_{i1} \rrbracket_{\mathcal{M}} \\ \quad | \dots \\ \quad | \text{out}(T_{ik}, \mathcal{M}(T_{ik})) \Rightarrow \llbracket \text{Nodes}_{ik} \rrbracket_{\mathcal{M}} \end{array} \right.$$

where  $\llbracket T_{i1} \rrbracket_{\mathcal{M}}$  involves the function extracted from the logical inductive type upon which  $T_{i1}$  is built. If  $T_{i1} = d_j\ t_1 \dots t_{p_j}$  then  $\llbracket T_{i1} \rrbracket_{\mathcal{M}} = f_{d_j}\ \text{in}(T_{i1}, \mathcal{M}(T_{i1}))$ .

Using the code generation algorithm described above, we obtain the following function from the binary search tree example introduced in Section 2:

$$\begin{aligned} & \text{fix } \text{search12 } (p1 : \text{bst}) (p2 : \text{nat}) : \text{path} := \\ & \quad \text{match } (p1, p2) \\ & \quad | (\text{Empty}, \_) \Rightarrow \text{Not\_found} \\ & \quad | (\text{Node } t1\ m\ t2, n) \Rightarrow \\ & \quad \quad (\text{match } \text{compare12 } n\ m \text{ with} \\ & \quad \quad | \text{Inf} \Rightarrow (\text{match } \text{search12 } t1\ p2 \text{ with } b \Rightarrow \text{Left } b) \\ & \quad \quad | \text{Sup} \Rightarrow (\text{match } \text{search12 } t2\ p2 \text{ with } b \Rightarrow \text{Right } b) \\ & \quad \quad | \text{Eq} \Rightarrow \text{End\_path}) \end{aligned}$$

### 3.4 Extensions of the Code Generation

This section proposes two extensions of the code generation algorithm. The first one concerns a larger family of non-deterministic specifications, while the second one aims to deal with full modes and incomplete specifications.

**Non-deterministic Specifications.** It is possible to accept specifications with overlapping conclusions where the premises cannot help distinguish between them, but can be ordered using an order over the patterns defined as follows:

**Definition 3 (Pattern Order).** *Given two patterns  $t_1$  and  $t_2$ ,  $t_1$  is more general than  $t_2$ , denoted by  $t_1 > t_2$ , iff the following property is verified:*

$$\begin{aligned}
 t_1 > t_2 &\Leftrightarrow (t_1 = v \wedge t_2 = c_l t_1 \dots t_{p_l}) \vee \\
 &(t_1 = \_ \wedge t_2 = c_l t_1 \dots t_{p_l}) \vee \\
 &(t_1 = c_l t'_1 \dots t'_{p_l} \wedge t_2 = c_l t_1 \dots t_{p_l} \wedge \\
 &\quad \exists i \in 1 \dots p_l, t'_i > t_i \wedge \forall i \in 1 \dots p_l, t'_i > t_i \vee t'_i = t_i)
 \end{aligned}$$

It should be noted that in the generated code, some decisions are made according to this order, and completeness may therefore be lost, i.e. some possible outputs w.r.t. the specification cannot be computed by the extracted function.

To illustrate this extension of the code generation algorithm, let us consider an improvement of the binary search tree example introduced in Section 2, which consists in adding two constructors (in bold font) in the *search* logical inductive type in order to correctly propagate the *Not\_found* value as follows:

**Inductive** *search* : *bst*  $\rightarrow$  *nat*  $\rightarrow$  *path*  $\rightarrow$  **Prop** :=  
 | *Search\_empty* : **forall** *n*, *search Empty n Not\_found*  
 | *Search\_found* : **forall** *n m t1 t2*, *compare n m Eq*  $\rightarrow$   
     *search (Node t1 m t2) n End\_path*  
 | *Search\_inf* : **forall** *n m t1 t2 b*, *search t1 n b*  $\rightarrow$   
     *compare n m Inf*  $\rightarrow$  *search (Node t1 m t2) n (Left b)*  
 | ***Searchinf\_nf*** : **forall** *n m t1 t2*, *search t1 n Not\_found*  $\rightarrow$   
     *compare n m Inf*  $\rightarrow$  *search (Node t1 m t2) n Not\_found*  
 | *Search\_sup* : **forall** *n m t1 t2 b*, *search t2 n b*  $\rightarrow$   
     *compare n m Sup*  $\rightarrow$  *search (Node t1 m t2) n (Right b)*  
 | ***Searchsup\_nf*** : **forall** *n m t1 t2*, *search t2 n Not\_found*  $\rightarrow$   
     *compare n m Sup*  $\rightarrow$  *search (Node t1 m t2) n Not\_found*.

As can be observed, the conclusions of *Search\_inf* and *Searchinf\_nf* overlap, and the two premises *search t1 n b* and *search t1 n Not\_found* as well, but these premises can be ordered: *b* is more general than *Not\_found*.

Considering this new kind of specifications requires some modifications in the rel-tree representation and consequently in the three related properties. The rel-tree representation is adapted to allow this ordering, and a rel-tree is defined by using lists of nodes instead of sets of nodes.

**Definition 4 (Rel-Tree with Lists).** *Given a logical inductive type  $d$ , it can be represented using the new following definition of rel-tree:*

$$\text{Rel-Tree}([(d\ t_{11} \dots t_{1p}, \text{Nodes}_1), \dots, (d\ t_{k1} \dots t_{kp}, \text{Nodes}_k)])$$

where  $\text{Nodes}_i$  is  $[(T_{i1}, \text{Nodes}_{i1}), \dots, (T_{ik}, \text{Nodes}_{ik})]$  or  $d\ t_{i1} \dots t_{ip}$ .

The previous properties SC, MCA, and NO must be adapted according to this new definition of rel-trees. In particular, in the property NO, we have to change the “not unifiable” statements by a new relation defined as follows:

**Definition 5 (Pattern Relation).** *Given two patterns  $t_1$  and  $t_2$ ,  $t_1$  is more general than  $t_2$  or non-unifiable with  $t_2$ , denoted by  $t_1 \succ t_2$ , iff the following property is verified:*

$$t_1 \succ t_2 \Leftrightarrow t_1 > t_2 \text{ or } t_1 \text{ and } t_2 \text{ are not unifiable}$$



As for the code generation algorithm from a rel-tree, it remains unchanged. Regarding the new specification of the binary search tree, the extracted function is then obtained as follows:

$$\begin{aligned}
 \text{fix search12 } (p1 : \text{bst}) (p2 : \text{nat}) : \text{path} := & \\
 \text{match } (p1, p2) & \\
 | (\text{Empty}, \_) \Rightarrow \text{Not\_found} & \\
 | (\text{Node } t1 \ m \ t2, n) \Rightarrow & \\
 (\text{match compare12 } n \ m \ \text{with} & \\
 | \text{Inf} \Rightarrow & \\
 (\text{match search12 } t1 \ n \ \text{with} & \\
 | \text{Not\_found} \Rightarrow \text{Not\_found} & \\
 | b \Rightarrow \text{Left } b) & \\
 | \text{Sup} \Rightarrow & \\
 (\text{match search12 } t2 \ n \ \text{with} & \\
 | \text{Not\_found} \Rightarrow \text{Not\_found} & \\
 | b \Rightarrow \text{Right } b) & \\
 | \text{Eq} \Rightarrow \text{End\_path}) &
 \end{aligned}$$

**Full Modes and Incomplete Specifications.** In the code generation described previously, we only consider partial modes and complete specifications. We are also able to deal with full modes and incomplete specifications. In the case of a full mode, the output of the extracted function is a boolean: *true* when the relation between the arguments is verified, *false* otherwise. The code generation follows the same algorithm than previously. In addition, for each case where a constructor is not defined, we add to the generated function the default case “|  $\_ \rightarrow \text{false}$ ”. Regarding incomplete specifications, if we follow the previous code generation algorithm, it produces a partially defined function, which is not supported by the CIC type theory. The algorithm has therefore to be adapted to extract a function whose result type is an option type of  $T$ , where  $T$  is the type of the output. The code generation then follows the same algorithm than previously, but for each case where a constructor is not defined, we add to the generated function the default case “|  $\_ \rightarrow \text{None}$ ”, where *None* is the empty constructor of the option type.

## 4 Soundness Proof Generation

In the previous section, we have explained how to extract functions from logical inductive types. In addition, we want to automatically provide proofs of soundness for these functions. In the following, we will only consider proofs of soundness for extractions of complete specifications with partial modes. However, the principle of soundness proof generation can be generalized to the other cases. The theorem of soundness has the following form:

$$\forall p_1, \dots, p_n, f_d p_1 \dots p_{n-1} = p_n \rightarrow d p_1 \dots p_n$$

where  $f_d$  is the name of the extracted function from the logical inductive type  $d$  with the mode  $\{1, \dots, n-1\}$ .

We prove the previous theorem by automatically providing a Coq proof script, which performs a functional induction using the extracted function [11]. Actually, for any function, Coq generates a functional induction scheme, which follows precisely the execution paths of the function (see the scheme `search12_ind` generated for the function `search12` in Section 2). When applying the induction scheme to the goal representing the theorem of soundness to be proved, we get a subgoal for each execution path of the extracted function. It should be noted that in the code generation described previously, we only use a high-level pattern matching, which is automatically compiled into a low-level pattern matching. This compilation may introduce some code duplication, and some “*let-in*” constructs are introduced to avoid the duplication of recursive calls. In the following, we will consider extracted functions where this compilation will have been performed, and which will be allowed to involve “*let-in*” expressions.

#### 4.1 Annotated Execution Paths

Before generating the proof script, we compute, from the generated code, the annotated execution paths, which correspond to the different cases of the functional induction scheme. An execution path is very similar to the target language used for the code generation, but it contains only one branch for each pattern matching. In the following,  $C$  will refer to the name of a constructor of a logical inductive type, while  $c$  will refer to a constructor of an inductive data type.

**Definition 6 (Annotated Execution Path).** *An annotated execution path is defined as follows:*

$$b ::= t \mid \text{let}_l x := t \text{ in } b \\ \mid \text{match } t \text{ with } c x_1 \dots x_p \Rightarrow_l b$$

where  $t$  is a term and  $l$  is an annotation which is either a set of constructor names  $\{C_1, \dots, C_n\}$ , or a set of premise positions  $\{(C_1, i_1), \dots, (C_n, i_n)\}$ , in which  $(C_i, i_k)$  denotes the  $i_k^{\text{th}}$  premise of constructor  $C_i$ .

This representation will help us generate the proof script because it contains information on both the generated code (and therefore the subgoal) and the specification (through the annotations). An annotation indicates the parts of the specification from whence the generated code comes. Thus, if  $(C, j)$  appears in the annotation  $l$  of a matching clause “ $c x_1 \dots x_p \Rightarrow_l b$ ”, then the constructor  $c$  appears in the premise  $P(C, j)$ . These annotated execution paths are generated from the extracted code, which is also annotated. The code generation algorithm is adapted to produce the annotations, which are initially computed from the specification and embedded in the rel-tree representation.

## 4.2 Proof Script of the Soundness Proof

As seen previously, we know that applying the functional induction scheme generates one subgoal per execution path of the extracted function. Each execution path is associated with one constructor in the specification, that we call  $C$ . One or more execution paths may be associated with the same constructor (due to the compilation of pattern matchings). Finally, it should be noted that we have an association between a constructor, an execution path, a case in the induction scheme, and a subgoal of the proof (once the induction scheme has been applied).

Each subgoal has the following form:

$$\forall \vec{v}, \forall \vec{v}_1, \vec{a}_1 \rightarrow H_1 \rightarrow \dots \forall \vec{v}_k, \vec{a}_k \rightarrow H_k \rightarrow \dots \forall \vec{v}_j, \vec{a}_j \rightarrow H_j \rightarrow d \ t_1 \dots t_p$$

where  $\vec{v}$ ,  $\vec{v}_k$ , with  $k$  in  $1 \dots j$ , are lists of variables,  $j$  the number of premises of the associated constructor  $C$ ,  $\vec{a}_k$  a list of equalities or “*let-in*” expressions corresponding to the associated annotated execution path, and  $H_k$  the soundness hypothesis if the logical inductive type involved in the premise,  $d_k$ , is  $d$  or an equality of the form  $f_k \ w_1 \dots w_u = r$ , where  $f_k$  is the extracted function for  $d_k$ . We assume that the extraction has been already performed from  $d_k$ , and as a consequence, the theorem of soundness related to  $f_k$  and  $d_k$  is available.

Each subgoal is transformed by applying successive introductions and rewritings using the several  $a_{ki}$ , leading to a goal which is the conclusion of the constructor  $C$  (up to renaming), and where each premise is present in the context. When  $a_{ki}$  is a “*let-in*” expression, it is transformed into an equality, and a rewriting is performed. The annotated execution paths are used to determine from which premise the equalities and “*let-in*” expressions of the subgoal come from. Thanks to this information, we know how to rewrite the goal. Finally, we apply the constructor  $C$  and this nearly finishes the proof of the subgoal: the arguments of the constructor are either present in the hypothesis context, or must be proved using the theorems of soundness related to the logical inductive types (other than  $d$ ) used in the constructor (like in the example of Section 2).

## 5 Implementation

We have implemented the extraction of logical inductive types within the Coq proof assistant as a plugin (not yet distributed, but available on demand by sending a mail to the authors). For information, another plugin (distributed since Coq version 8.4) allows the user to extract ML code from logical inductive types. In the short term, we plan to merge these two plugins.

With the current implementation, it is possible to extract specifications involving several logical inductive types, but there are some restrictions. First, the definitions must not be mutually recursive, and the extracted functions must rely on structural recursion. Moreover, logical inductive types must contain neither logical connectives ( $\wedge$ ,  $\vee$ , or  $\neg$ ), nor equality symbols. Finally, regarding proofs of soundness, we are only able to generate them for complete functions extracted with partial modes. These restrictions should be relaxed in the near future.

## 6 Conclusion

We have presented an operational approach allowing the extraction of computational content written as a Coq function from a Coq inductive specification. This extracted function is accompanied by a proof of soundness establishing that the result of the function complies with the specification. Future work will consist in completing the proof generation: generating soundness proof for the other kinds of modes and specifications, and also (when it is relevant) producing completeness proofs. The former is just an adaptation of the approach presented here (i.e. a functional induction exploiting annotations produced during the code generation), while the latter requires a different proof generation scheme. The next step will be to address inductive specifications embedding a general recursion. Finally, we could also try to extract functions from non-terminating specifications (e.g. the semantics of a language featuring a while loop), expressed as mixed inductive-coinductive definitions (see [8] for some examples). A simple approach would consist in adding to the extracted function a non-negative integer counter which bounds the depth of the computation (as done in CompCert [5]).

## References

1. Barthe, G., Courtieu, P.: Efficient Reasoning about Executable Specifications in Coq. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 31–46. Springer, Heidelberg (2002)
2. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning Inductive into Equational Specifications. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 131–146. Springer, Heidelberg (2009)
3. Berghofer, S., Nipkow, T.: Executing Higher Order Logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 24–40. Springer, Heidelberg (2002)
4. Bertot, Y., Capretta, V., Das Barman, K.: Type-Theoretic Functional Semantics. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 83–98. Springer, Heidelberg (2002)
5. Blazy, S., Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning (JAR)* 43(3), 263–288 (2009)
6. Delahaye, D., Dubois, C., Étienne, J.-F.: Extracting Purely Functional Contents from Logical Inductive Types. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 70–85. Springer, Heidelberg (2007)
7. Le Fessant, F., Maranget, L.: Optimizing Pattern-Matching. In: Pierce, B.C. (ed.) International Conference on Functional Programming (ICFP). SIGPLAN, pp. 26–37. ACM (2001)
8. Leroy, X., Grall, H.: Coinductive Big-Step Operational Semantics. *Information and Computation (IC)* 207(2), 284–304 (2009)
9. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
10. The Coq Development Team. Coq, version 8.4. INRIA (August 2012), <http://coq.inria.fr/>

# The New Quickcheck for Isabelle

## Random, Exhaustive and Symbolic Testing under One Roof

Lukas Bulwahn

Institut für Informatik, Technische Universität München, Germany

**Abstract.** The new Quickcheck is a counterexample generator for Isabelle/HOL that uncovers faulty specifications and invalid conjectures using various testing strategies. The previous Quickcheck only tested conjectures by random testing. The new Quickcheck extends the previous one and integrates two novel testing strategies: exhaustive testing with concrete values; and symbolic testing, evaluating conjectures with a narrowing strategy. Orthogonally to the strategies, we address two general issues: First, we extend the class of executable conjectures and specifications, and second, we present techniques to deal with conditional conjectures, i.e., conjectures with premises. We evaluate the testing strategies and techniques on a number of specifications, functional data structures and a hotel key card system.

## 1 Introduction

Counterexample generators are very useful advisory tools for users of interactive theorem provers. They make developing and proving specifications an enjoyable experience. Users can identify errors leading to invalid conjectures by immediate counterexamples rather than by time-consuming unsuccessful proof attempts.

Isabelle [13] uncovers invalid conjectures by two means: Refute [17] and Nitpick [3] search for countermodels by reducing a conjecture to boolean satisfiability, whereas Quickcheck *tests* a conjecture by assigning values to the free variables of the conjecture and evaluating it. To evaluate the conjecture efficiently, Quickcheck translates the conjecture and related definitions to an ML or Haskell program, exploiting Isabelle’s code generation infrastructure [10]. This allows Quickcheck to test a conjecture with millions of test cases within seconds.

In earlier work [1], Quickcheck was originally modeled after the QuickCheck tool for Haskell [7], which tests user-supplied properties of a Haskell program with randomly generated values. The first contribution of this work is to extend Quickcheck with exhaustive and narrowing-based testing as complements to random testing. Exhaustive testing checks the formula for every possible set of values up to a given bound, and hence finds counterexamples that random testing might miss. Narrowing-based testing evaluates the formula symbolically rather than with a finite set of ground values, and therefore, it can be more precise and more efficient than the other two approaches.

Another contribution is to address previous weaknesses of counterexample generation by testing. Quickcheck is inherently limited to *executable* specifications, and consequently the specification must be transformed into a functional program. We extend the class of executable conjectures in several directions:

- Narrowing-based testing can handle unbounded existential quantifiers over infinite types, enabling refutation for a class of conjectures where all other counterexample generators fail due to their imprecision or lack of support.
- For polymorphic conjectures, Quickcheck finds counterexamples by evaluating the conjecture for all finite models of small sizes.
- Quickcheck now handles *underspecified functions*, and provides a simple user interface to cope with arbitrary type definitions.

A well-known problem of testing with concrete values are *conditional conjectures*, especially those with very restrictive premises. These conjectures are problematic because when testing naively, for the vast majority of variable assignments the premise is not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the premise into account when generating values. We present three solutions for Quickcheck to generate only appropriate variable assignments:

- Derivation of custom test data generators from user declarations
- Automatic synthesis of test data generators that take the condition’s definition into account
- Symbolic evaluation

To measure the impact of our improvements, we compare the various testing approaches in Quickcheck on a large set of automatically generated conjectures, on faulty implementations of functional data structures, and on a formalization of a hotel key card system, which was until now beyond the reach of Isabelle’s counterexample generators. In a unrefereed paper [2], we previously presented an overview of this work as one part of Isabelle’s latest developments.

The paper is structured as follows. We begin with Quickcheck’s basic infrastructure (§2 and §3) for testing with concrete values, i.e., random and exhaustive testing. We show how to deal with conditional conjectures (§4) to avoid the vacuous test cases that plague most specification testing tools. Then we discuss the advantages of narrowing-based testing (§5). We highlight aspects (§6) that improve Quickcheck’s performance and complete its infrastructure. Our evaluation (§7) sheds further light on the counterexample generators’ strength.

## 2 From Conjectures to Test Programs

Given a conjecture, Quickcheck builds a test program that combines the conjecture’s evaluation with the generation of test values. This test program is then passed to Isabelle’s code generator, which executes it efficiently within Isabelle’s underlying ML runtime system. Turning the conjecture into a test program is a step common to both random and exhaustive testing.

We create a test program for a given conjecture by enclosing its evaluation with test data generators for its free variables. The test program returns the counterexample as an optional value: It either returns *Some x*, where *x* is a counterexample, or *None*. Both testing approaches define test data generators. A generator creates a finite domain of values and performs a test for a given conjecture to all elements of that domain. Our presentation here focuses on exhaustive testing. The construction for random testing is analogous.

Given a function *c* that checks the conjecture for a single value, the generator *exhaustive c* yields a function that checks the conjecture for all values up to a given bound. For feedback for the user, a counterexample of type  $\tau$  is mapped to a fixed type *result* using the function *reify* ::  $\tau \Rightarrow \text{result}$ . We describe the generators in detail in §3. A simple test program for a conjecture *C* with a single variable *x* can be expressed as:

$$\text{exhaustive } (\lambda x. \text{ if } C \ x \ \text{then } \text{None} \ \text{else } \text{Some } (\text{reify } x))$$

Test programs are improved by taking the common structure of conjectures into account, as a list of premises and a conclusion. If a premise does not depend on a free variable, the generation of values for this free variable can be postponed until after checking the premise. Thus, Quickcheck optimizes the test program so that it generates the values for each variable as late as possible.

For example, consider the function *insert*, which inserts an element into a sorted list in such a way that it remains sorted. If *insert* is implemented correctly, the following property should hold:

$$\text{sorted } xs \implies \text{sorted } (\text{insert } x \ xs)$$

Quickcheck generates values for *xs* and checks the premise *sorted xs*. Now only for values fulfilling the premise, Quickcheck proceeds generating values for *x*, and checks the conclusion *insert x xs*. Consequently, Quickcheck produces this optimized test program:

$$\begin{aligned} \text{exhaustive } (\lambda xs. \text{ if } \neg \text{sorted } xs \ \text{then } \text{None} \\ \text{else } \text{exhaustive } (\lambda x. \text{ if } \text{sorted } (\text{insert } x \ xs) \ \text{then } \text{None} \\ \text{else } \text{Some } (\text{reify } (x, \ xs)))) \end{aligned}$$

In the presence of (multiple) premises, this interleaving of generation and evaluation already improves its performance dramatically. In §4, we optimize the generation and evaluation of this kind of conjectures even more.

### 3 Test Data Generators

Quickcheck automatically synthesizes test data generators for random and exhaustive testing (§3.1 and §3.2). For both testing strategies, Quickcheck supports the definition of generators: Generators of inductive data types (§3.3) are automatically defined, and generators of arbitrary type definitions (§3.4) can be defined with some guidance from the user.

Both testing approaches build on a family of test data generators. These test data generators are type-based, i.e., there is exactly one generator for each type. Generators for a complex type  $\tau$  are constructed following its type structure, which is nicely described using type classes in Isabelle [18]. For example, given a generator for polymorphic lists  $\alpha$  *list* and a generator for the type of natural numbers (type *nat*), the generator for *nat list* is implicitly composed from those two generators by the type class mechanism. Throughout the presentation, we denote an instance of an overloaded constant  $c$  with type  $\tau$  by  $c_\tau$ .

Generators are put together by *chaining* and *choosing between alternatives*. The generators express a nondeterministic (branching) computation. The generators' operations are closely related to operations on a *plus monad*, a generalization of the ideas for nondeterministic computations in [16].

### 3.1 Basic Random Generators

Random generators are provided by the type class *random*, which defines a function *random* of type  $\text{nat} \Rightarrow \text{seed} \Rightarrow \tau \times \text{seed}$  for type  $\tau$  in this class. The generator yields a value of type  $\tau$ , and is parametrized by the size of values to be generated. The state *seed* is used for the underlying random engine. Random generators are chained together by the *return* and *bind* (written infix as  $\gg=$ ) operators on an open state monad:

$$\begin{aligned} \text{return} &:: \alpha \Rightarrow \sigma \Rightarrow \alpha \times \sigma \\ \text{return } x \ s &= (x, s) \\ \gg= &:: (\sigma \Rightarrow \alpha \times \sigma) \Rightarrow (\alpha \Rightarrow \sigma \Rightarrow \beta \times \sigma) \Rightarrow \sigma \Rightarrow \beta \times \sigma \\ (f \gg= g) \ s &= g \ x \ s' \ \text{where } (x, s') = f \ s \end{aligned}$$

With this notation, the random generator for product types is built from generators for its type constructor's arguments, where  $i$  denotes the size:

$$\text{random}_{\alpha \times \beta} \ i = \text{random}_\alpha \ i \gg= (\lambda x. \text{random}_\beta \ i \gg= (\lambda y. \text{return} \ (x, y)))$$

Given a list of generators with associated weights, *select* yields a random generator that chooses one of the generators (randomly using the *seed* value). The weights are used to give a non-uniform probability distribution to the alternatives. The random generator for the sum type  $\alpha + \beta$  (with constructors *Inl* and *Inr*) illustrates selecting of alternative generators:

$$\text{random}_{\alpha + \beta} \ i = \text{select} \ [(1, \text{random}_\alpha \ i \gg= (\lambda x. \text{return} \ (\text{Inl} \ x))), \\ (1, \text{random}_\beta \ i \gg= (\lambda x. \text{return} \ (\text{Inr} \ x)))]$$

### 3.2 Basic Exhaustive Generators

Similar to random generators, exhaustive generators are provided by the type class *exhaustive* with a function *exhaustive* of type  $(\tau \Rightarrow \text{result option}) \Rightarrow \text{nat} \Rightarrow \text{result option}$ . The exhaustive generators are expressed with continuations: They take a continuation (which ultimately checks the conjecture), and evaluate it with



all values of type  $\tau$  up to the given size. Generators are chained by nesting the continuations. For example, for a given continuation  $c$  and size  $i$ , the generator for product types is defined by

$$\mathit{exhaustive}_{\alpha \times \beta} c i = \mathit{exhaustive}_{\alpha} (\lambda x. \mathit{exhaustive}_{\beta} (\lambda y. c (x, y))) i i$$

As the weights of alternatives are irrelevant for exhaustive testing, generators can be simply combined with the binary operation  $\sqcup$ , which chooses the first *Some* value when evaluating from left to right:

$$\begin{aligned} \sqcup &:: \alpha \textit{ option} \Rightarrow \alpha \textit{ option} \Rightarrow \alpha \textit{ option} \\ (\textit{Some } x) \sqcup y &= \textit{Some } x \\ \textit{None} \sqcup y &= y \end{aligned}$$

The generator for  $\alpha + \beta$  joins the two exhaustive generators for types  $\alpha$  and  $\beta$  employing the operator  $\sqcup$ :

$$\begin{aligned} \mathit{exhaustive}_{\alpha + \beta} c i &= \\ &\mathit{exhaustive}_{\alpha} (\lambda x. c (\textit{Inl } x)) i \sqcup \mathit{exhaustive}_{\beta} (\lambda x. c (\textit{Inr } x)) i \end{aligned}$$

### 3.3 Generators for Inductive Datatypes

Most commonly, new types are defined by datatype declarations. For these types, Quickcheck automatically constructs random and exhaustive generators upon the type's definition. The construction of random generators has been described in [11], so we only sketch the construction of exhaustive generators here.

We view a datatype as a recursive type definition of a sum of product types. For example, the datatype  $\alpha \textit{ list}$  can be seen as least fixed point of the equation  $\alpha \textit{ list} = \textit{unit} + \alpha \times (\alpha \textit{ list})$ . Following the scheme of exhaustive generators for product and sum type, the exhaustive generator for lists is defined recursively:

$$\begin{aligned} \mathit{exhaustive}_{\alpha \textit{ list}} c i &= \textit{if } (i = 0) \textit{ then } \textit{None} \textit{ else } (c \textit{ Nil} \sqcup \\ &\mathit{exhaustive}_{\alpha} (\lambda x. \mathit{exhaustive}_{\alpha \textit{ list}} (\lambda xs. c (\textit{Cons } x xs)) (i - 1)) i) \end{aligned}$$

Generalizing this example to an arbitrary datatype is almost straightforward, only recursion through functions takes some care.

### 3.4 Generators for Arbitrary Type Definitions

Beyond inductive datatypes, types can also be defined by other means, e.g., by HOL-style type definitions. For such types, code generation requires special setup by the user. Quickcheck provides a simple interface with which users can specify generators. One simply lists the *constructing functions* for values of this type. Generators are then built using these functions, as if they were datatype constructors for this type. For example, red-black trees are binary search trees with a sophisticated invariant. The type  $(\alpha, \beta) \textit{ rbt}$  contains all binary search trees with keys of type  $\alpha$  and values of type  $\beta$  fulfilling the invariant. Values of this type can be generated with the invariant-preserving operations:

$$\begin{aligned} \text{empty} &:: (\alpha, \beta) \text{ rbt} \\ \text{insert} &:: \alpha \Rightarrow \beta \Rightarrow (\alpha, \beta) \text{ rbt} \Rightarrow (\alpha, \beta) \text{ rbt} \end{aligned}$$

Using these as constructing functions, Quickcheck provides random and exhaustive generators for  $(\alpha, \beta) \text{ rbt}$  that produce values starting with the *empty* tree and executing a sequence of *insert* operations. The random generator chooses the key and value for the *insert* operation randomly from the set of possible values, whereas the exhaustive generator enumerates all possible keys and values (up to a given size) for the *insert* operations.

## 4 Conditional Conjectures

The main weakness of both random and exhaustive testing, already mentioned in the original QuickCheck for Haskell paper, is that they do not cope well with hard-to-satisfy premises. For example, when testing our previous conjecture about *insert*,

$$\text{sorted } xs \Longrightarrow \text{sorted } (\text{insert } x \text{ } xs)$$

the conjecture is evaluated with all lists up to a given bound for *xs*. For all unsorted lists, the premise is not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the condition into account when generating values: In this example, we would like to only generate sorted lists.

Often, these conditional conjectures arise in the verification of functional data structures, e.g., red-black trees. A properly implemented *delete* operation for red-black trees satisfies the following property:

$$\text{is-rbt } t \Longrightarrow \text{is-rbt } (\text{delete } k \text{ } t)$$

The predicate *is-rbt* captures the invariant of red-black trees on the type of binary search trees  $(\alpha, \beta) \text{ tree}$ . Again, binary trees generated naively rarely satisfy the premise, and we prefer to only generate trees satisfying the invariant.

### 4.1 Custom Generators

The simplest solution to test conditional conjectures effectively is to employ a custom generator that has been provided by the user. Assuming the user provides a generator for some type restricted by a predicate (cf. §3.4) that matches the condition, Quickcheck automatically *lifts* the conjecture to the restricted type. For example, the conjecture about *delete* is automatically lifted to the type  $(\alpha, \beta) \text{ rbt}$ , where  $\text{Rep}_{\text{rbt}} t'$  maps a red-black tree  $t'$  of type  $(\alpha, \beta) \text{ rbt}$  to its representative binary tree on type  $(\alpha, \beta) \text{ tree}$ :

$$\text{is-rbt } (\text{Rep}_{\text{rbt}} t') \Longrightarrow \text{is-rbt } (\text{delete } k \text{ } (\text{Rep}_{\text{rbt}} t'))$$

Note that  $t'$  is now of type  $(\alpha, \beta) \text{ rbt}$ , unlike the original conjecture, where  $t$  has the type  $(\alpha, \beta) \text{ tree}$ . As all representatives of type  $(\alpha, \beta) \text{ rbt}$  satisfy the predicate *is-rbt* (by the type's construction), the premise *is-rbt*  $(\text{Rep}_{\text{rbt}} t')$  simplifies to *true*. This way, Quickcheck obtains an unconditional conjecture, which it tests either with the random or exhaustive generator of  $(\alpha, \beta) \text{ rbt}$ .

## 4.2 Smart Generators

A more sophisticated solution to test conditional conjectures effectively, is smart test data generators that take the condition's definition into account. These test data generators construct values in a bottom-up fashion, simultaneously testing the condition and generating appropriate values.

For our conjecture about *insort*, Quickcheck can automatically derive a test data generator that only constructs *sorted* lists. From the definition for *sorted*,

$$\begin{aligned} \text{sorted Nil} &= \text{True} \\ \text{sorted } [x] &= \text{True} \\ \text{sorted } (x_1 \cdot (x_2 \cdot xs)) &= (x_1 \leq x_2 \wedge \text{sorted } (x_2 \cdot xs)), \end{aligned}$$

we obtain an exhaustive generator that constructs sorted lists choosing either *Nil*, a singleton list  $[x]$ , or appending an element to the front of a sorted list if the element is smaller than the list's head:

$$\begin{aligned} \text{exhaustive-sorted}_{\alpha \text{ list } c} i &= \text{if } (i = 0) \text{ then None else } ((c \text{ Nil}) \sqcup \\ &(\text{exhaustive}_{\alpha} (\lambda x. c [x]) (i - 1)) \sqcup \\ &(\text{exhaustive-sorted}_{\alpha \text{ list}} (\lambda xs'. \text{case } xs' \text{ of Nil} \Rightarrow \text{None} \\ &| x_2 \cdot xs \Rightarrow \text{exhaustive}_{\alpha} (\lambda x_1. \text{if } (x_1 \leq x_2) \text{ then } c (x_1 \cdot (x_2 \cdot xs)) \\ &\text{else None})) (i - 1)) (i - 1)) \end{aligned}$$

Briefly, we synthesize these generators by reformulating the definitions as a set of Horn clauses and computing its data-flow dependencies (cf. [4] for more details). Applying these generators, Quickcheck's performance improves significantly (cf. §7.2).

## 5 Narrowing-Based Testing

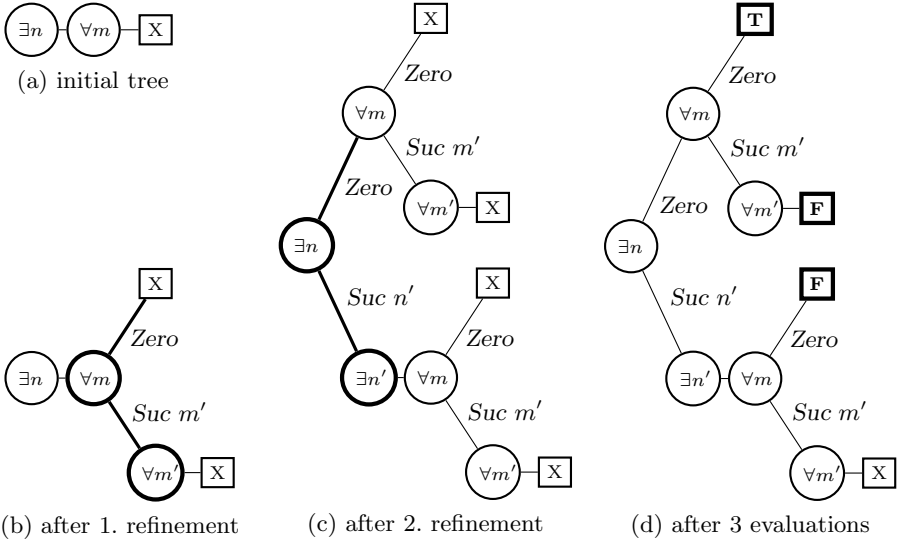
The random and exhaustive strategies suffer from two important limitations: They cannot refute propositions that existentially quantify over infinite types, and they often repeatedly test formulas with values that checks essentially the same executions (e.g., because of symmetries).

Both issues arise from the use of ground values and can be addressed by evaluating the formula symbolically. The technique is called narrowing and is well known from term rewriting. The main idea is to evaluate the conjecture with partially instantiated terms and to progressively refine these terms as needed. The following simple conjecture illustrates the benefit of the narrowing approach.

$$\exists n :: \text{nat}. \forall m :: \text{nat}. n = m$$

To disprove it, we must show for every natural number  $n$  that  $\exists m. n \neq m$ . Taking a symbolic view, if  $n = 0$ , we can choose any  $m \neq 0$  and if  $n > 0$ , then 0 can serve as a witness for  $m$ .

At its core, the mechanism evaluates boolean expressions where free variables are substituted by partially instantiated terms. These terms are *constructor terms*, i.e., they are built from datatype constructors and distinct variables,



**Fig. 1.** Refinement tree for the evaluation of  $\exists n :: nat. \forall m :: nat. m = n$

e.g., *Suc*  $n$ , *Zero* and *Cons*  $x_1$  (*Cons*  $x_2$   $x_3$ ). Exploiting evaluations in Haskell, an expression with partial terms is evaluated to head normal form as far as possible: The execution either returns the (ground) head normal form if it is reduced despite variables in the initial term, or it indicates which variable is critical for the evaluation. For the evaluation of a boolean expression, it yields ground values *true* or *false*, if the expression is true or false for all substitutions of the free variables, resp., or it indicates the critical variable. For example, the execution determines that  $Zero \neq Suc\ n$  is true for all natural numbers  $n$ , but the value of  $Suc\ Zero \neq Suc\ n$  depends the value of  $n$ .

On top of this evaluation for partial terms, there is a refinement algorithm that refutes formulas in prenex normal form. It uses a *refinement tree* that records the results of the evaluation with partial terms and keeps track of refinements. The tree is used to determine the formula’s truth value and successive evaluations with partial terms. Figure 1 shows the refinement tree during the refutation of the conjecture  $\exists n :: nat. \forall m :: nat. m = n$ .

Leaves of the tree carry the evaluation’s result: initially *unknown* (X), and after the evaluation, the definite results *true* (T) or *false* (F). Inner nodes carry a variable and are classified as universal or existential. Branches are annotated with simple substitutions for its parent’s variable, i.e., variables are assigned a single constructor with fresh variables as arguments. A path from the root to a leaf represents an assignment of partial terms by composing the substitutions along the path. For example, the path to the node annotated with T in Fig. 1d assigns  $n$  and  $m$  to *Zero*.

The truth value of a tree is defined recursively: The leafs' values are given by their annotations; the value of a universal node is the conjunction of the values of its subtrees; dually for existential nodes, it is the disjunction of its subtrees. Conjunction and disjunction are defined as in Kleene's three-valued logic with *unknown* as the third value. Starting with an initial tree with no refinements, the refinement algorithm does the following three steps:

1. Find by depth-first search a leaf, that makes the tree's truth value *unknown*, and evaluate the property with the partial terms associated with this leaf.
2. If the evaluation yields a boolean truth value, the leaf is annotated. If the evaluation calls for a refinement, the refinement tree is altered reflecting a case distinction on the critical variable.
3. If the new tree's truth value is false, we have found a counterexample to the conjecture. If it remains *unknown*, we continue with the first step. If the evaluation requires too many refinement steps, the execution is aborted. In rare cases, the tree's truth value might be true, proving the conjecture.

The illustrated evaluation in Fig. 1 starts with an initial tree that represents the quantifier part of the formula above and one leaf annotated with  $X$  (Fig. 1a). The first evaluation of  $m = n$  with symbolic values  $m$  and  $n$  leads the tool to refine  $m$ . The top-most constructor of  $m$  can either be *Zero* or *Suc* (Fig. 1b). The next evaluation with  $m \mapsto \text{Zero}$  requires a refinement of  $n$ , resulting in the state of Fig. 1c. Now, the evaluation with  $n \mapsto \text{Zero}, m \mapsto \text{Zero}$  yields true, and for  $n \mapsto \text{Zero}, m \mapsto \text{Suc } m_1$  with some fresh variable  $m_1$  yields false. As the truth value of the upper branch  $n \mapsto \text{Zero}$  is false, we continue with the lower branch  $n \mapsto \text{Suc } n'$ . The last evaluation for  $n \mapsto \text{Suc } n_1, m \mapsto \text{Zero}$  yields false, and thus shows the invalidity of the formula (Fig. 1d). We note that the refutation never evaluated  $n \mapsto \text{Suc } n_1, m \mapsto \text{Suc } m_1$ .

The above example is perhaps too simple to be convincing. A more realistic example is based on the observation that the palindrome  $[a, b, b, a]$  can be split into the list  $[a, b]$  and its reverse  $[b, a]$ . Generalizing this to arbitrary lists, we boldly conjecture that

$$\text{rev } xs = xs \implies \exists ys. xs = ys @ \text{rev } ys$$

The narrowing approach immediately finds the counterexample  $xs = [a_1]$ , inferring that there is no witness for  $ys$  in the infinite domain of lists: If  $ys$  is empty,  $ys @ \text{rev } ys = [] \neq [a_1]$ , and if  $ys$  is not empty,  $ys @ \text{rev } ys$  consists of at least two elements and hence cannot be equal to  $[a_1]$ .

Narrowing also deals very well with conditional conjectures. In our example with the *delete* operation on red-black trees,

$$\text{is-rbt } t \implies \text{is-rbt } (\text{delete } k \ t)$$

the premise *is-rbt*  $t$  ensures that the tree  $t$  has a black root node, and in fact, after a few refinements, narrowing will only test symbolic values satisfying this property, already pruning away about half of the overall test cases.

## 6 Completing the Infrastructure

So far, we presented the core parts of Quickcheck. In the following subsections, we touch on two further aspects: testing of polymorphic conjectures and underspecified functions.

### 6.1 Polymorphic Conjectures

If the conjecture is polymorphic, we can instantiate the type variables with any concrete type for refuting it. Older versions of Quickcheck instantiated type variables with the type of integers (if possible depending on the type class constraints), and tested the conjecture with increasing integer values. Lately, Quickcheck prefers to use a set of small finite types instead, so that conjectures with quantifiers, e.g., existential conjectures  $\exists x :: \alpha. P x$ , can be refuted by a finite number of  $P$  tests.

The implementation for refuting quantified formulas over a finite type is based on the type class *enum*. This allows us to obtain implementations for more complex types by composition. E.g., the type  $\alpha \times \beta \Rightarrow \gamma$  is finite if  $\alpha$ ,  $\beta$  and  $\gamma$  are finite types. The type class *enum* provides three operations for every finite type  $\tau$ : *univ* ::  $\tau$  *list* enumerates the finite universe; *all* ::  $(\tau \Rightarrow \text{bool}) \Rightarrow \text{bool}$  and *ex* ::  $(\tau \Rightarrow \text{bool}) \Rightarrow \text{bool}$  check universal and existential properties. The existential and universal quantifiers could be expressed just with *univ* ::  $\tau$  *list*, i.e.,  $\forall x :: \tau. P x = \text{list-all } P (\text{univ} :: \tau \text{ list})$ . Due to the strict evaluation of ML, this would be rather inefficient: The evaluation would first construct a finite (but potentially large) list of values, and then check them sequentially. To avoid the large intermediate list, we implement the quantifiers using continuations, similar to the construction of the exhaustive generators (cf. §3.2). For example, the universal quantifiers for product and sum type are implemented by

$$\begin{aligned} \text{all}_{\alpha \times \beta} P &= \text{all}_{\alpha} (\lambda a :: \alpha. \text{all}_{\beta} (\lambda b :: \beta. P (a, b))) \\ \text{all}_{\alpha + \beta} P &= (\text{all}_{\alpha} (\lambda a :: \alpha. P (\text{Inl } a))) \wedge \text{all}_{\beta} (\lambda b :: \beta. P (\text{Inr } b)) \end{aligned}$$

For most types, the implementation is straightforward. Only for the function type, it is a bit more involved. To construct the set of all functions  $\alpha \Rightarrow \beta$ , we have to create all possible mappings, i.e., all lists of type  $\beta$  *list* with the same length as *univ* ::  $\alpha$  *list*, and transform those lists into functions.

### 6.2 Underspecified Functions

Even though HOL is a logic of total functions, users can give underspecified function definitions. The results are total functions, but equations only exist for some subset of possible inputs. A prominent example here is the head function on lists. It is specified by  $hd (\text{Cons } x \ xs) = x$ , but no equation is given for the *Nil* constructor. Some facts only hold on the domain where the function is specified, while others may hold in general, even on values where the function has no specifying equations. For example, the conjecture about *hd* and *append*,

$$hd (append\ xs\ ys) = (if\ xs = Nil\ then\ hd\ ys\ else\ hd\ xs),$$

is valid for all lists  $xs$  and  $ys$ , even if  $xs$  and  $ys$  are  $Nil$ . In this special case, left and right hand side are equal, i.e., they reduce to the same term  $hd\ Nil$ . In contrast, the conjecture  $hd (map\ f\ xs) = f (hd\ xs)$  is valid only if  $xs \neq Nil$ . In the presence of underspecified function definitions, Quickcheck cannot distinguish the two cases occurring in the examples above. In other words, it cannot determine if a counterexample in the examples above is genuine or spurious. Therefore, it marks the counterexample as *potentially spurious*. On the two conjectures above, Quickcheck returns the potentially spurious counterexamples  $xs = Nil, ys = Nil$  and  $xs = Nil, f = \lambda x. a_1$ . Nevertheless, these potentially spurious counterexamples are useful in two ways: First, it makes users aware that the choice of how the underspecified function is turned into a total function might be crucial for the validity of this conjecture; second, when users know that the property only holds on values where the function is properly specified, they can validate that the given assumptions suffice to restrict the values to the defined part of the function by observing that no potentially spurious counterexample is found.

To uncover counterexamples with underspecified functions, we slightly change the test programs. The evaluation of underspecified functions in Standard ML yields a `Match` exception if it encounters a call to such a function and no pattern matches the given arguments. The test program catches this exception. If we are interested in possible counterexamples due to underspecification, Quickcheck returns the values that yield the exception as counterexample. Alternatively, if we are only interested in genuine counterexamples, Quickcheck continues to search for other values.

## 7 Empirical Results

We evaluated Quickcheck with its different strategies on a database of theorem mutations, faulty implementations of functional data structures, and a trace-based hotel key card system<sup>[1]</sup>. The functional data structures and the key card system are well suited for comparing the different techniques to cope with conditional conjectures.

### 7.1 Evaluation on Theorem Mutations

To obtain a large set of non-theorems in Isabelle, we derive formulas *mutating* existing theorems by replacing constants and swapping arguments, as in [1,3]. Table 1 shows the results of running the counterexample generators on 400 mutated theorems of 13 theories with a very liberal time limit of 30 seconds. The chosen set of theories focuses on executable ones, and leaves out those that are obviously not executable. For example, theories with axiomatic definitions or

<sup>1</sup> The test data is available at <http://www21.in.tum.de/~bulwahn/cpp2012.tar.gz>

**Table 1.** Results for running counterexample generators on mutated theorems on a Intel Core2 Duo T7700 2.40GHz with a time limit of 30 seconds

Theory	Counterexample generators			Nitpick
	Random	Exhaustive	Narrowing	
<b>Arithmetics</b>				
Divides [fin]	199/318	212/318	221/343	259/400
Divides [int]	224/369	239/369	248/394	
GCD	203/294	203/294	228/336	216/400
MacLaurin [fin]	44/61	44/61	45/77	19/400
MacLaurin [int]	55/79	55/79	56/95	
<b>Set Theory</b>				
Fun [fin]	214/394	215/394	201/396	235/400
Fun [int]	146/254	144/254	161/326	
Relation [fin]	248/395	251/395	248/395	247/400
Relation [int]	139/230	155/230	160/258	
Set [fin]	246/395	246/395	249/395	260/400
Set [int]	205/329	206/329	220/369	
Wellfounded [fin]	229/372	233/372	232/373	249/400
Wellfounded [int]	45/94	47/94	51/122	
<b>Datatypes</b>				
List [fin]	197/319	197/318	215/354	245/400
List [int]	191/312	193/312	212/351	
Map [fin]	257/400	257/400	257/400	258/400
Map [int]	146/221	148/221	160/248	
<b>AFP Theories</b>				
Huffman	244/399	248/399	246/399	251/400
List-Index	256/399	256/399	263/399	271/400
Max-Card-Matching [fin]	152/345	212/345	212/345	214/400
Max-Card-Matching [int]	4/11	4/11	4/11	
Regular-Sets	154/304	152/304	210/368	142/400

coinductive datatypes are not executable with Isabelle’s code generation. Conjectures in these theories are only refuted by Nitpick.

The four columns show the absolute number of genuine counterexamples of the different approaches: random testing, exhaustive testing, narrowing-based testing, and Nitpick. In a cell with values A/B, A is the number of genuine counterexamples and B the number of executable mutants of the corresponding counterexample generator. As Nitpick handles arbitrary specifications, it is able to check all 400 mutants. Quickcheck can use finite types or integers to instantiate polymorphic conjectures (cf. §6.1). For theories with polymorphic conjectures, we show both modes separately in the table, indicated with [fin] and [int]. Using finite types for polymorphic conjectures makes almost all conjectures in the set theory domain amenable to Quickcheck, closing the previously existing gap between Quickcheck and Nitpick in this domain. The narrowing-based testing



can execute more conjectures than concrete testing with random and exhaustive testing. We gain most on the Regular-Sets theory, increasing from 304 to 368.

We also compared the tools against each other, and measured the number of counterexamples that can be found uniquely by one tool compared to another. Exhaustive testing slightly outperforms random testing. Narrowing often finds a few more counterexamples than exhaustive testing, but this is mainly due to the larger set of executable formulas. Narrowing and Nitpick complement each other to some extent, as witnessed most prominently by Isabelle’s GCD theory. In absolute numbers, narrowing and Nitpick find 228 and 216 counterexamples; hence only differing by 12. However, they succeed on different conjectures—narrowing finds 23 counterexamples where Nitpick fails, Nitpick finds 11 where narrowing fails—meaning that employing them in combination yields 239 counterexamples.

To illustrate the differences in strength between testing with Quickcheck and model finding with Nitpick, we show two interesting examples of our evaluation. On the one hand, consider one of the monotonicity lemmas for integer division:

$$b \cdot q + r = b' \cdot q' + r' \wedge 0 \leq b' \cdot q' + r' \wedge r' < b' \wedge 0 \leq r \wedge 0 < b' \wedge b' \leq b \implies q \leq q'$$

For Quickcheck, it is no problem to detect two typos that change the second premise to  $0 \leq b' \cdot b' + r'$  and the fifth premise to  $0 < q'$ . It produces the counterexample  $b = -2$ ;  $q = 3$ ;  $r = 1$ ;  $b' = -2$ ;  $q' = 1$ ;  $r' = -3$  instantaneously, while Nitpick replies after seven minutes with a similar counterexample.

On the other hand, in the Isabelle theory of maximal matchings in graphs (Max-Card-Matching), a certain invalid conjecture is refuted by constructing a graph with 4 vertices and a matching with two edges. Owing to the power of its SAT solver, Nitpick finds this matching within a few seconds. Exhaustive testing tries to enumerate all graphs and searches for matchings quite naively. Thus, Quickcheck needs roughly a minute to find a counterexample. Random testing does not find the counterexample, even with 100,000 iterations for each size and testing a few minutes—a matching for a valid graph is too unlikely to obtain by randomly chosen values. Narrowing prunes the search space before evaluating the conjecture with all possible concrete values, and finds a counterexample in about thirty seconds.

These two examples demonstrate the strength of both tools: Quickcheck is strong on arithmetics, while Nitpick handles well boolean constraints over finite domains.

## 7.2 Functional Data Structures

Beyond the mutations of lemmas, we evaluated the different testing approaches on faulty implementations of typical functional data structures. We injected faults by adding typos into the correct implementations of the delete operation of AVL trees, red-black trees, and 2-3 trees. By adding typos, we create 10 different (possibly incorrect) versions of the delete operation for each data structure. On 2-3 trees, we check two invariants of the delete operation, keeping the tree balanced and ordered, i.e.,  $balanced\ t \implies balanced\ (delete\ k\ t)$ , and  $ordered\ t \implies ordered\ (delete\ k\ t)$ . We check two similar properties for AVL

**Table 2.** Number of counterexamples on faulty implementations of functional data structures (time limit: 30 s for AVL and red-black trees; 120 s for 2-3 trees)

	R <sub>2K</sub>	R <sub>20K</sub>	Exh.	Cu.G.	Sm.G.	Nar.	Nit.
AVL trees	5	7	7	9	9	11	4
Red-black trees	10	18	21	22	19	26	11
2-3 trees	5	5	7	11	12	12	0

trees, and three similar properties for red-black trees. With the 10 versions, this yields 20 tests each for 2-3 and AVL trees, and 30 tests for red-black trees, on which we apply various counterexample generators. In this setting, we compare the techniques to deal with *conditional conjectures*. Random testing is applied with 2,000 and 20,000 iterations for each size (abbrev. R<sub>2K</sub>, R<sub>20K</sub>). Furthermore, we used exhaustive testing (Exh.), custom generators (Cu.G., §4.1), smart generators (Sm.G., §4.2), narrowing (Nar.) and Nitpick (Nit.).

Table 2 summarizes the results. Overall, narrowing, smart, and custom generators beat exhaustive testing, which itself performs better than random testing and Nitpick. Nitpick struggles with large functional programs and is limited to shallow errors in the smaller implementations of AVL and red-black trees. Increasing the number of iterations for random testing helps, but in our experience, it does not find any error that was not also found by testing exhaustively. For the 2-3 trees, the smart generators and narrowing find errors in 5 more cases than exhaustive testing. However, in principle, exhaustive testing should find the errors eventually. Thus, in these more intrinsic cases, we increased the time for the naive exhaustive testing to finally discover the fault. However, even after one hour of testing, exhaustive testing was not able to detect a single one of them. This shows that using the test data generators and narrowing-based testing in this setting is clearly superior to naive exhaustive testing. The smart generators and narrowing find 12 errors in 20 conjectures. In the eight cases where they did not find anything within the time limit, even testing more thoroughly for an hour did not reveal any further errors. Most probably, the property still holds, as the randomly injected faults do not necessarily affect the invariant.

### 7.3 Trace-Based Hotel Key Card System

As a further case study, we checked a hotel key card system by Nipkow [12]. The faulty system contains a tricky man-in-the-middle attack, which is only uncovered by a trace of length 6. The formalization uses a restrictive predicate that describes in which order specific events occur. Due to the occurrence of existential quantifiers, the original specification is not executable for random and exhaustive testing. Even after refinements to obtain an executable reformulation, the naive random and exhaustive testing fail to find the counterexample within ten minutes of testing, as the search space is too large. Smart generators include some processing that detects if the values of existential quantifiers are

bound in the formula. Therefore, we do not have to reformulate the specification when we use smart generators. Employing these smart generators, we can find the attack within a few seconds. Narrowing can handle the existential quantifiers in principle, but in practice it performs badly with the deeply nested existential quantifiers in the specification. This renders it impossible to find the counterexample with narrowing. After eliminating the existential quantifiers manually, we also obtain a counterexample with narrowing within a few seconds.

On this trace-based version of the hotel key card system, Nitpick fails to find the counterexample with a time limit of ten minutes. However, Nitpick finds the counterexample on an equivalent *state-based* formalization of the hotel key card system (cf. [3], §6.2). This indicates that Quickcheck and Nitpick excel on formalizations with different specification styles: Nitpick on relational descriptions, Quickcheck on realistic functional programs and trace-based descriptions.

## 8 Related Work

The success story of Haskell’s QuickCheck [7] has led to many descendants in interactive theorem provers. Besides Isabelle, PVS [14], Agda [8], ACL2 [9] and ACL2 Sedan [5] include a random testing tool like the original QuickCheck.

The tool in ACL2 Sedan simplifies the conjecture using a synergistic combination of random testing and theorem proving: The application of selected theorem proving methods before testing can ease testing of conditional conjectures.

Our exhaustive testing is inspired by Haskell’s SmallCheck [15], but is targeting ML with its strict evaluation. The implementation of Haskell’s SmallCheck takes advantage of its laziness, simplifying the definition of generators, while Isabelle’s tool takes the strictness of ML into account and uses continuations.

Tools using narrowing for testing functional programs symbolically are the Apsy tool [11] for Agda, EasyCheck [6] for the programming language Curry, and LazySmallCheck [15] for Haskell. Like LazySmallCheck, our implementation exploits Haskell’s lazy evaluation and *imprecise exceptions*. The refinement algorithm of [15] only allows universal properties whereas our refinement algorithm can also deal with existential quantifiers.

## 9 Conclusion

As we have seen, the methods to uncover invalid conjectures, testing and model finding, implemented by the counterexample generators Quickcheck and Nitpick in Isabelle, have their justification. Quickcheck with its new testing strategies and our effort to extend its applicability allows to check many conjectures effectively that were previously beyond the scope of testing. Isabelle’s users benefit from having all these strategies at their disposal, because they complement each other very well. Unmentioned so far, Quickcheck’s performance also profits from the fact that code generation in Isabelle is becoming more common and widely used. Isabelle’s library provides many additions to set up code generation for numerous

purposes. To validate specifications with simple examples before proving, users invest some time to make their specifications executable. Quickcheck returns this investment the first time users encounter an invalid conjecture, so they can correct an error immediately instead of wasting hours on an impossible proof.

**Acknowledgment.** I thank Jasmin Blanchette, Brian Huffman, Peter Lamich, Tobias Nipkow, Lars Noschinski, Andrei Popescu, Thomas Tuerk, Dmitry Traytel, Tjark Weber and the anonymous referees for suggesting several textual improvements. I acknowledge funding from DFG doctorate program 1480 (PUMA).

## References

1. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) SEFM 2004, pp. 230–239. IEEE C.S. (2004)
2. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic Proof and Disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
4. Bulwahn, L.: Smart Testing of Functional Programs in Isabelle. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 153–167. Springer, Heidelberg (2012)
5. Chamathi, H.R., Dillinger, P., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving (2011), <http://arxiv.org/pdf/1105.4394>
6. Christiansen, J., Fischer, S.: EasyCheck — Test Data for Free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 322–336. Springer, Heidelberg (2008)
7. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP 2000, pp. 268–279. ACM (2000)
8. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining Testing and Proving in Dependent Type Theory. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 188–203. Springer, Heidelberg (2003)
9. Eastlund, C.: Doublecheck your theorems. In: 8th Int. Workshop on the ACL2 Theorem Prover and its Applications (2009)
10. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
11. Lindblad, F.: Property directed generation of first-order test data. In: Morazán, M. (ed.) TFP 2007, pp. 105–123. Intellect (2008)
12. Nipkow, T.: Verifying a Hotel Key Card System. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 1–14. Springer, Heidelberg (2006)
13. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

14. Owre, S.: Random testing in PVS. In: AFM 2006 (2006)
15. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In: Haskell Symp. 2008, pp. 37–48 (2008)
16. Wadler, P.: How to Replace Failure by a List of Successes. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985)
17. Weber, T.: SAT-based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Institut für Informatik, Technische Universität München, Germany (2008)
18. Wenzel, M.: Type Classes and Overloading in Higher-order Logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997)

# Proving Concurrent Noninterference<sup>\*</sup>

Andrei Popescu<sup>1,2</sup>, Johannes Hölzl<sup>1</sup>, and Tobias Nipkow<sup>1</sup>

<sup>1</sup> Technische Universität München

<sup>2</sup> Institute of Mathematics Simion Stoilow, Romania

**Abstract.** We perform a formal analysis of compositionality techniques for proving possibilistic noninterference for a while language with parallel composition. We develop a uniform framework where we express a wide range of noninterference variants from the literature and compare them w.r.t. their *contracts*: the strength of the security properties they ensure weighed against the harshness of the syntactic conditions they enforce. This results in a simple implementable algorithm for proving that a program has a specific noninterference property, using only compositionality, which captures uniformly several security type-system results from the literature and suggests a further improved type system. All formalism and theorems have been mechanically verified in Isabelle/HOL.

## 1 Introduction

*Language-based noninterference* is an important and well-studied security property. To state this property, one assumes the program memory is separated into a *low*, or public, part, which an attacker is able to observe, and a *high*, or private, part, hidden to the attacker. Then a program satisfies noninterference if, upon running it, the high part of the initial memory does not affect the low part of the resulting memory. Thus, the program has *no information leaks* from the private part of the memory into the public one, so that a potential attacker should not be able to obtain information about private data by inspecting public data.

Noninterference comes in several different variants, depending on what type of channels one accepts as capable of transmitting leaks—besides the normal channels represented by program variables, so-called *covert channels* include termination and timing channels. Moreover, when nondeterminism is involved, one can distinguish between *possibilistic* and *probabilistic* noninterference (the latter also taking probabilistic channels into account).

In this paper, we deal with noninterference in the presence of *possibilistic concurrency*. The literature abounds in notions of concurrent possibilistic noninterference and techniques to enforce it [2–6, 14, 19–21, 24, 29, 31], many of them surveyed in [23]. There is usually a tradeoff between the strength of a security property and the harshness of the conditions imposed on the programs in order to satisfy it (typically, a type system). Yet, new methods for establishing noninterference are often presented as improvements over older methods (e.g., a more lenient type system) while being rather

---

<sup>\*</sup> Supported by the DFG project Ni 491/13–1 (part of the DFG priority program RS3) and the DFG RTG 1480.

brief on the notion that in effect the whole *contract* is being changed: less pressure on the programs, weaker noninterference ensured.

The paper presents the first comparison of a variety of noninterference notions and results, in a unified and formalized framework, where complex results from the literature are given uniform and simplified proofs. As a preview of the kind of properties we analyze and classify in this paper, here is a selection of informal notions of a command  $c$  being secure (noninterfering):

(1) Given any two initial memory states that are indistinguishable by the attacker (have the same low, i.e., public, part), the executions of  $c$  proceed identically w.r.t. both the program counter and the updates on the low part of the memory—we call this property *self isomorphism*.

(2)  $c$  may never change the low part of the memory during its execution—we call this *discreetness* (often in the literature this is called *highness*).

(3) If started in two indistinguishable memory states, the executions of  $c$  are lock-step bisimilar, performing the same updates to the low part of the memory—we call this *self strong bisimilarity*, i.e., strong bisimilarity to itself (called *strong security* in [25]).

(4) A relaxation of strong bisimilarity with lock-step synchronization replaced by *01-bisimilarity* (simply called *bisimilarity* in [5]), where only attacker-visible (i.e., low-memory changing) steps in one execution are required to be matched by corresponding steps in the other, while “discreet” (i.e., low-memory unchanging) steps need not be matched. Thus, one step may be matched by either zero or one steps.

(5) A further relaxation of strong bisimilarity—*weak bisimilarity* [16] (used in [4,29] in a security context) where one step may be matched by any number of steps.

Property 1 (self isomorphism) is a very strong security notion, ensuring that an attacker controlling the low inputs of  $c$  is not able to infer any information about the high inputs, not even if he is allowed to observe the low part of intermediate memory states *and the program counter*. In particular, self isomorphism exhibits no leaks on covert channels such as timing or termination. Property 2 (discreetness) is neither weaker nor stronger than self isomorphism, but it no longer guarantees indistinguishability w.r.t. the program counter, and moreover the attacker may infer confidential information by measuring execution time. Property 3 (strong bisimilarity) prevents leaks on standard channels (low variable values) and timing channels, but, unlike self isomorphism, does not guarantee that execution starting in indistinguishable states follow the very same paths (taking the same branches). Properties 4 (01-bisimilarity) and 5 (weak bisimilarity) are weakenings of all of the above three. They are only able to guarantee the absence of leakage through standard channels.

**Example 1.** Consider the following commands, where  $l$  is a low variable and  $h, h'$  are high variables:

- $c_0: h := 0$
- $c_1: \text{if } l = 0 \text{ then } h := 1 \text{ else } l := 2$
- $c_2: \text{if } h = 0 \text{ then } h := 1 \text{ else } h := 2$
- $c_3: \text{if } h = 0 \text{ then } h := 1 ; h := 2 \text{ else } h := 3$
- $c_4: l := 4 ; c_3$
- $c_5: c_3 ; l := 4$

- $c_6$ :  $l := h$
- $c_7$ :  $h' := 0$  ; while  $h > 0$  do  $\{h := h - 1 ; h' := h' + 1\}$  ;  $l := 4$

$c_0$  is both self isomorphic and discreet.  $c_1$  is self isomorphic (since it is not testing any high variable), but not discreet.  $c_2$  and  $c_3$  are discreet (as they are not updating any low variable), but not self isomorphic.  $c_1$  and  $c_2$ , but not  $c_3$ , are self strongly bisimilar—the reason why  $c_3$  is not is its branching on a high test in conjunction with one branch taking longer than the other.  $c_4$  is self 01-bisimilar, because, after a self isomorphic assignment, it transits to a discreet continuation.  $c_5$  is not self 01-bisimilar, but it is self weakly bisimilar.  $c_6$  is not secure according to any of the five criteria—it exhibits a direct leak from high to low.

If we ignore timing channels and assume that initially  $h \geq 0$ , then it is reasonable to consider  $c_7$  secure, since it has the same effect as the program  $h' := h ; l := 4$ . However, whether or not we should deem  $c_7$  secure *when placed in parallel with other threads* depends on the assumption we make on these threads—e.g., are they allowed to change  $h$ , thus preventing termination of  $c_7$ ?

Note that the above example programs are sequential, which seems to contrast with our declared focus on concurrency—the explanation, hinted in the previous paragraph and detailed throughout the paper, is that the discussed notions of noninterference are defined anticipating parallel composition, i.e., so that the subject threads behave well when placed in parallel with other threads.

Here is an overview of this paper, where we use “security” and “noninterference” as synonyms. We start by introducing the concurrent setting where we operate: a while language with parallel composition and a fixed attacker-indistinguishability relation on program states (§2). Then we systematize and compare bisimilarity-based notions from the literature (§3). A formal study of the compositionality of, and of the implications between, these notions (§4) yields a novel proof methodology: To show that  $c$  is secure according to some notion  $N$ , first try to reduce the goal to proving  $N$  for the components of  $c$ ; if this is not feasible due to failure of the required compositionality of  $N$  w.r.t. the language construct  $Cns$  located at the top of  $c$  (e.g.,  $Cns$  can be an *If*, or a *While*, etc.), try to identify a stronger notion  $M$  that is (more) compositional w.r.t.  $Cns$ , and so on, recursively. The compositionality caveats of existing notions suggests the definition of a fully compositional security notion (§5). We then look at existing work on security type systems in the light of our analysis (§6)—the aforementioned simple proof technique turns out quite insightful, capturing these type system criteria uniformly. Our novel security notion from §5 yields a more permissive syntactic criterion than the existing ones, but the result targets only terminating programs. Finally, we discuss end-to-end security aspects of the studied bisimilarity-based notions (§7). We do not present any proofs of the stated facts—however, a (readable) Isabelle formalization of this paper’s development, together with a map connecting the formal scripts with the propositions stated in this paper, is available at [18].

## 2 The Programming Language

We consider a simple while language with parallel composition, whose set **com** of commands, ranged over by  $c, d, e$ , is given by the following grammar:



$$c ::= atm \mid \text{Seq } c_1 c_2 \mid \text{If } tst c_1 c_2 \mid \text{While } tst c \mid \text{Par } c_1 c_2$$

Above,  $atm$  ranges over an unspecified set **atom** of atomic commands (atoms). Standard examples of atoms are assignments such as  $x := x + y$ .  $\text{Seq } c_1 c_2$  is the sequential composition of  $c_1$  and  $c_2$ , written in concrete syntax as  $c_1 ; c_2$ .  $\text{If } tst c_1 c_2$  is the conditional, written in concrete syntax as  $\text{if } tst \text{ then } c_1 \text{ else } c_2$ , where  $tst$  ranges over an unspecified set **test** of tests. Standard examples of tests are Boolean expressions such as  $x = y$ .  $\text{While } tst c$  is the usual while loop, in concrete syntax,  $\text{while } tst \text{ do } c$ .  $\text{Par } c_1 c_2$  is the parallel composition of  $c_1$  and  $c_2$ , in concrete syntax,  $c_1 \parallel c_2$ . We generally prefer abstract syntax in theoretical results and concrete syntax in examples.

To give semantics to the language, we assume: a set of (memory) states, **state**, ranged over by  $s, t$ ; an execution function for the atoms,  $\text{aexec} : \mathbf{atom} \rightarrow \mathbf{state} \rightarrow \mathbf{state}$ ; an evaluation function for the tests,  $\text{tval} : \mathbf{test} \rightarrow \mathbf{state} \rightarrow \mathbf{bool}$ . Then we define a standard small-step semantics [17] as a pair of inductive predicates  $\rightarrow_{\tau} : (\mathbf{com} \times \mathbf{state}) \rightarrow \mathbf{state}$  and  $\rightarrow_c : (\mathbf{com} \times \mathbf{state}) \rightarrow (\mathbf{com} \times \mathbf{state})$  (where the subscripts T and C stand for “termination” and “continuation”) specified in Fig. 1. Intuitively, we interpret  $(c, s) \rightarrow_{\tau} s'$  as stating: in state  $s$ , command  $c$  may take a step terminating while changing the state to  $s'$ ; and  $(c, s) \rightarrow_c (c', s')$  as saying: in state  $s$ , command  $c$  may take a step yielding the continuation  $c'$  while changing the state to  $s'$ . The pairs  $(c, s)$ , which we call *configurations*, are thus thought of as consisting of the part of the program that remains to be executed,  $c$ , and the current state,  $s$ . We carefully distinguish between continuation and terminating steps (as the two predicates  $\rightarrow_c$  and  $\rightarrow_{\tau}$ ), since termination-sensitiveness will be crucial in our development.  $\rightarrow_c^*$  denotes the reflexive-transitive closure of  $\rightarrow_c$ , and  $\rightarrow_{\tau}^*$  the composition of  $\rightarrow_c^*$  with  $\rightarrow_{\tau}$ . Thus,  $(c, s) \rightarrow_c^* (c', s')$  means that  $(c', s')$  is reachable from  $(c, s)$  by zero or more continuation steps, and  $(c, s) \rightarrow_{\tau}^* s'$  that (the final state)  $s'$  is reachable from  $(c, s)$  by zero or more continuation steps followed by a terminating step.

$$\begin{array}{c}
(atm, s) \rightarrow_{\tau} \text{aexec } atm \ s \\
\hline
\text{tval } tst \ s \\
\hline
(\text{If } tst \ c_1 \ c_2, s) \rightarrow_c (c_1, s) \\
\hline
(c_1, s) \rightarrow_c (c'_1, s') \\
\hline
(\text{Par } c_1 \ c_2, s) \rightarrow_c (\text{Par } c'_1 \ c_2, s') \\
\hline
(c_2, s) \rightarrow_c (c'_2, s') \\
\hline
(\text{Par } c_1 \ c_2, s) \rightarrow_c (\text{Par } c_1 \ c'_2, s')
\end{array}
\qquad
\begin{array}{c}
(c_1, s) \rightarrow_{\tau} s' \\
\hline
(\text{Seq } c_1 \ c_2, s) \rightarrow_c (c_2, s') \\
\hline
\neg \text{tval } tst \ s \\
\hline
(\text{If } tst \ c_1 \ c_2, s) \rightarrow_c (c_2, s) \\
\hline
(c_2, s) \rightarrow_{\tau} s' \\
\hline
(\text{Par } c_1 \ c_2, s) \rightarrow_c (c_1, s') \\
\hline
(c_2, s) \rightarrow_{\tau} s' \\
\hline
(\text{Par } c_1 \ c_2, s) \rightarrow_c (c_1, s')
\end{array}
\qquad
\begin{array}{c}
(c_1, s) \rightarrow_c (c'_1, s') \\
\hline
(\text{Seq } c_1 \ c_2, s) \rightarrow_c (\text{Seq } c'_1 \ c_2, s') \\
\hline
\neg \text{tval } tst \ s \\
\hline
(\text{While } tst \ c, s) \rightarrow_{\tau} s \\
\hline
\text{tval } tst \ s \\
\hline
(\text{While } tst \ c, s) \rightarrow_c (\text{Seq } c \ (\text{While } tst \ c), s) \\
\hline
(c_1, s) \rightarrow_{\tau} s' \\
\hline
(\text{Par } c_1 \ c_2, s) \rightarrow_c (c_2, s')
\end{array}$$

Fig. 1. Small-step semantics

### 3 Notions of Noninterference

Next we proceed to a uniform description of several notions of noninterference from the literature. We fix a relation  $\sim$  on states, called *indistinguishability*, where  $s \sim t$  is meant to say “ $s$  and  $t$  are indistinguishable by the attacker.”

**Example 2.** Often,  $\sim$  is defined as follows. We assume that atomic statements and tests are built by means of arithmetic and boolean expressions applied to variables taken from a set **var**. States are assignments of values to variables, i.e., the set **state** is  $\mathbf{var} \rightarrow \mathbf{val}$ , where **val** is a set of values (e.g., integers). Variables are classified as either low (lo) or high (hi) by a given security level function  $\text{sec} : \mathbf{var} \rightarrow \{\text{lo}, \text{hi}\}$ . Then  $\sim$  is defined as coincidence on the low variables, with the intuition that the attacker is only able to observe these. Formally,  $s \sim t \equiv \forall x \in \mathbf{var}. \text{sec } x = \text{lo} \implies s x = t x$ .

We define the following predicates on commands *coinductively as greatest fixed points*, i.e., as the *strongest* predicates satisfying the indicated clauses:

- *Self isomorphism*, *siso*, by  $\text{siso } c \equiv$ 

$$(\forall s t c' s'. s \sim t \wedge (c, s) \rightarrow_c (c', s') \implies (\exists t'. (c, t) \rightarrow_c (c', t') \wedge s' \sim t')) \wedge$$

$$(\forall s t s'. s \sim t \wedge (c, s) \rightarrow_\tau s' \implies (\exists t'. (c, t) \rightarrow_\tau t' \wedge s' \sim t')) \wedge$$

$$(\forall s c' s'. (c, s) \rightarrow_c (c', s') \implies \text{siso } c')$$
- *Discreetness*, *discr*, by  $\text{discr } c \equiv$ 

$$(\forall s c' s'. (c, s) \rightarrow_c (c', s') \implies s \sim s' \wedge \text{discr } c') \wedge (\forall s s'. (c, s) \rightarrow_\tau s' \implies s \sim s')$$

The coinductive definition of self isomorphism expresses that that execution of a command proceeds absolutely independently of the indistinguishability class of the state, and this is true *interactively*, i.e., *regardless of the intervention of the environment*, provided this intervention is itself compatible with the state indistinguishability relation. And similarly for the definition of *discr*, expressing that the command never changes the indistinguishability class, regardless of what that class has become due to potential action from the environment.

The last aspect, interactivity, is expressed by the universal quantification over the indistinguishable states  $s$  and  $t$  in the definition of *siso*. Indeed, even though transitions operate on (command, state) pairs, the *siso* predicate operates on commands alone, forgetting each time the result state  $s'$  from the continuation  $(c', s')$ . Thus, at each resumption point, the predicate quantifies universally over *all* states  $s$  (“overwriting” the previous  $s'$ ), to account for the fact that the new state produced by the command under consideration may have been changed by the environment (perhaps consisting of other threads running in parallel, and/or of the attacker) before that command gets to perform an other step. For example, the command  $c \equiv h := 0 ; l := h$  (with  $h$  high and  $l$  low) would be deemed as self isomorphic if it were not for the interactivity constraint. Indeed, if no interference from the environment is assumed, the execution of  $c$  proceeds the same way regardless of the initial value of  $h$ , as it first assigns 0 to  $h$ . However, *siso*  $c$  does not hold, since the continuation  $l := h$  is required to be secure *given any value of  $h$*  arising as the effect of a secure thread running in parallel, say,  $h := h'$  with  $h'$  high. This interactivity twist (originating from [22, 25]) is convenient for compositionality, since it ensures that a command is secure not only in isolation, but also if placed in any pool of secure threads running in parallel. As a consequence, most of the security notions discussed in this paper will be interactive.

We shall also need the following interactive notion of termination possibility at each point during execution, via the coinductively defined predicate  $\text{mayT}$  (read “may terminate”):  $\text{mayT } c \equiv \forall s c' s'. (c, s) \rightarrow_c (c', s') \implies (\exists s''. (c', s') \rightarrow_\tau^* s'') \wedge \text{mayT } c'$ .

Self isomorphism and discreteness were expressible as unary predicates. However, interesting noninterference properties may require binary relations. To see this, assume we wish to express that  $c$  is secure, i.e., its executions are (multi)step-wise equivalent if started in indistinguishable states. Assume  $c$  branches according to a high test. Then indistinguishable states may yield different continuations, say,  $c_1$  and  $c_2$ , and so we are faced with the problem of proving the executions of  $c_1$  and  $c_2$  (multi)step-wise equivalent, i.e., proving  $c_1$  and  $c_2$  *bisimilar*. (The above two notions have by-passed this problem in trivial ways: self isomorphism forbids this situation by disallowing the program counter to diverge, hence disallowing high tests, while discreteness of  $c$  also requires  $c_1$  and  $c_2$  to be discreet, hence trivially “equivalent”.)

$\text{match}_C^C \theta c d \equiv$ $\forall s t c' s'. s \sim t \wedge (c, s) \rightarrow_C (c', s') \implies$ $(\exists d' t'. (d, t) \rightarrow_C (d', t') \wedge s' \sim t' \wedge \theta c' d')$	
$\text{match}_{01C}^C \theta c d \equiv$ $\forall s t c' s'. s \sim t \wedge (c, s) \rightarrow_C (c', s') \implies$ $(\exists d' t'. (d, t) \rightarrow_C (d', t') \wedge s' \sim t' \wedge \theta c' d') \vee$ $(s' \sim t \wedge \theta c' d)$	$\text{match}_{MC}^C \theta c d \equiv$ $\forall s t c' s'. s \sim t \wedge (c, s) \rightarrow_C (c', s') \implies$ $(\exists d' t'. (d, t) \rightarrow_C^* (d', t') \wedge s' \sim t' \wedge \theta c' d')$
$\text{match}_{01}^C \theta c d \equiv$ $\forall s t c' s'. s \sim t \wedge (c, s) \rightarrow_C (c', s') \implies$ $(\exists d' t'. (d, t) \rightarrow_C (d', t') \wedge s' \sim t' \wedge \theta c' d') \vee$ $(s' \sim t \wedge \theta c' d) \vee$ $(\exists t'. (d, t) \rightarrow_T t' \wedge s' \sim t' \wedge \text{discr } c')$	$\text{match}_M^C \theta c d \equiv$ $\forall s t c' s'. s \sim t \wedge (c, s) \rightarrow_C (c', s') \implies$ $(\exists d' t'. (d, t) \rightarrow_C^* (d', t') \wedge s' \sim t' \wedge \theta c' d') \vee$ $(\exists t'. (d, t) \rightarrow_T^* t' \wedge s' \sim t' \wedge \text{discr } c')$
$\text{match}_T^T c d \equiv$ $\forall s t s'. s \sim t \wedge (c, s) \rightarrow_T s' \implies$ $(\exists t'. (d, t) \rightarrow_T t' \wedge s' \sim t')$	$\text{match}_{MT}^T c d \equiv$ $\forall s t s'. s \sim t \wedge (c, s) \rightarrow_T s' \implies$ $(\exists t'. (d, t) \rightarrow_T^* t' \wedge s' \sim t')$
$\text{match}_{01}^T c d \equiv$ $\forall s t s'. s \sim t \wedge (c, s) \rightarrow_T s' \implies$ $(\exists t'. (d, t) \rightarrow_T t' \wedge s' \sim t') \vee$ $(\exists d' t'. (d, t) \rightarrow_C (d', t') \wedge s' \sim t' \wedge \text{discr } d') \vee$ $(s' \sim t \wedge \text{discr } d)$	$\text{match}_M^T c d \equiv$ $\forall s t s'. s \sim t \wedge (c, s) \rightarrow_T s' \implies$ $(\exists t'. (d, t) \rightarrow_T^* t' \wedge s' \sim t') \vee$ $(\exists d' t'. (d, t) \rightarrow_C^* (d', t') \wedge s' \sim t' \wedge \text{discr } d')$

Fig. 2. Matchers

In order to define relevant notions of bisimilarity, it will be useful to first introduce matching operators (or *matchers*) that express various choices of rules for the bisimilarity game. They are defined in Fig. 2, where  $\theta$  ranges over binary relations on commands. In the operator names, the superscripts indicate the kind of steps being taken, and the subscripts indicate by what kind of steps these must be simulated (matched), where: “C” means (single) continuation step; “T” means (single) terminating step; “01C” means 0 or 1 continuation steps; “01” means 0 or 1 continuation or terminating steps, i.e., 01C or T; “MC” means multiple continuation steps; “MT” means multiple continuation steps, followed by a terminating step; “M” means MC or MT. E.g.,  $\text{match}_C^C$  refers to matching any continuation step by a continuation step,  $\text{match}_{01C}^C$  to matching any continuation step by 0 or 1 continuation steps, i.e., either by a continuation step or by a stutter move.

Matchers indicate how the single steps of a command  $c$  may be matched by *single or multiple* steps of a command  $d$ . In most cases, the matcher is also parameterized by a continuation relation  $\theta$ ; exceptions are  $\text{match}_T^\top$  and  $\text{match}_{MT}^\top$ , where, due to termination of both the left and the right sides, no continuation makes sense.  $\text{match}_{01}^c$ ,  $\text{match}_{01}^\top$ ,  $\text{match}_M^c$  and  $\text{match}_M^\top$  are termination-flexible matchers, in that they allow matching continuation steps against termination steps and vice versa. For instance,  $\text{match}_{01}^c$  (“match a continuation step against 0 or 1 steps of either kind”) requires for  $\theta$ ,  $c$  and  $d$  that, for all indistinguishable states  $s$  and  $t$ , any step  $(c, s) \rightarrow_c (c', s')$  be matched by either a continuation step  $(d, t) \rightarrow_c (d', t')$ , or a stutter step, or a termination step  $(d, t) \rightarrow_\tau t'$ . In each case, it is also required that the resulting states are indistinguishable. Moreover, in the first two cases (for continuation and stutter) it is required that the resulting commands are in relation  $\theta$ . For the third case though (the termination step), the latter condition does not make sense, since on the left of the matcher we have a continuation command,  $c'$ , while the right side has terminated; what we require instead is that, w.r.t. the attacker-observable behavior,  $c'$  acts as if it terminated, in that it will never change the indistinguishability class of the state, i.e., is discreet. (Similar discreetness conditions appear in the definitions of the other termination-flexible matchers for similar reasons.)

We are now ready to define the following bisimilarity relations, again coinductively, by plugging in different combinations of matchers and taking each time the *largest symmetric relation* satisfying the given clause (where the bisimilarities are written with infix notation on the left and are passed as arguments to the matchers on the right):

- *Strong bisimilarity*,  $\approx_s$ , by  $c \approx_s d \equiv \text{match}_c^c (\approx_s) c d \wedge \text{match}_T^\top c d$
- *01-bisimilarity*,  $\approx_{01}$ , by  $c \approx_{01} d \equiv \text{match}_{01}^c (\approx_{01}) c d \wedge \text{match}_{01}^\top c d$
- *Termination-sensitive 01-bisimilarity (OIT-bisimilarity)*,  $\approx_{01T}$ , by  $c \approx_{01T} d \equiv \text{match}_{01c}^c (\approx_{01T}) c d \wedge \text{match}_T^\top c d$
- *Weak bisimilarity*,  $\approx_w$ , by  $c \approx_w d \equiv \text{match}_M^c (\approx_w) c d \wedge \text{match}_M^\top c d$
- *Termination-sensitive weak bisimilarity (weak T-bisimilarity)*,  $\approx_{wT}$ , by  $c \approx_{wT} d \equiv \text{match}_{Mc}^c (\approx_w) c d \wedge \text{match}_{MT}^\top c d$

All these bisimilarity relations are by definition symmetric and can also be proved transitive, but they are *not* reflexive. In fact, the notion of a command  $c$  being bisimilar with itself (e.g.,  $c \approx_s c$ ,  $c \approx_{01} c$ , etc.), which we call *self bisimilarity of  $c$*  (e.g., self strong bisimilarity, self 01-bisimilarity, etc.) is taken in this paper as the formalization of the informal notion of security of a command. Below we explain how different bisimilarities correspond to different attacker models.

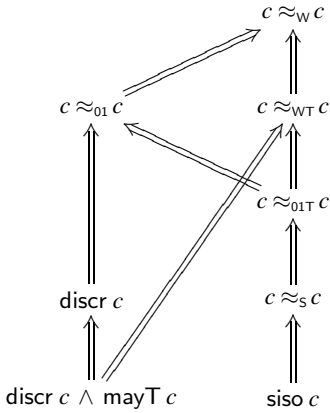
In all cases, one assumes the attacker has access to the program (command) source code and the low part of the state, and the ability to set, at the beginning of the command execution, the low part of the state in any desired way. For strong bisimilarity ( $\approx_s$ ), we assume the attacker’s ability to repeatedly stop the program after single execution steps and inspect the (low part of the) state, or, equivalently, take snapshots of the state after controlled numbers of execution steps. Technically, this shows in the two involved matchers,  $\text{match}_c^c$  and  $\text{match}_T^\top$ , being one-to-one (w.r.t. continuation or termination steps). Moreover, we assume the attacker can detect termination—this shows in the fact that the two matchers preserve the type of transition: continuation vs. continuation and termination vs. termination. For weak bisimilarity ( $\approx_w$ ), the attacker may still stop the program repeatedly, but has no control on the number of steps that the program takes

between two steps. (For what the attacker knows, zero, one, or more steps could have been taken.) This shows in the one-to-many nature of the matchers. The termination-sensitive version of weak bisimilarity ( $\approx_{wT}$ ) additionally assumes the attacker is able to detect termination. Thus,  $\approx_{wT}$  allows, via  $\text{match}_{wT}^\top$ , matching a termination step by a sequence of steps only if the latter ends in a termination step. 01-bisimilarity ( $\approx_{01}$ ), also coming with a termination-sensitive variant ( $\approx_{01T}$ ), is intermediate between strong and weak bisimilarity. Here, the attacker may keep running the program for 0 or 1 steps, without knowing which of the two situations has actually occurred.

The following proposition, relating different notions of self bisimilarity, follows easily from the definitions of the corresponding matchers:

**Prop 1.** The implications in Fig. 3 hold.

Note that discreteness implies self 01-bisimilarity, but not self 01T-bisimilarity. However, for may-terminating processes (roughly, processes with finite behavior), it does imply self wT-bisimilarity.



**Fig. 3.** Implications between security notions

$c$	$\text{mayT } c$	$\text{discr } c$	$\varphi c$	$\psi c$
$\text{atm}$	True	$\text{pres atm}$	$\text{cpt atm}$	$\text{cpt atm}$
Seq $c_1 c_2$	$\text{mayT } c_1$	$\text{discr } c_1$	$\varphi c_1$	$\psi_T c_1$
	$\text{mayT } c_2$	$\text{discr } c_2$	$\varphi c_2$	$\psi c_2$
If $\text{tst } c_1 c_2$	$\text{mayT } c_1$	$\text{discr } c_1$	$\text{cpt } \text{tst } \varphi c_1$	$\text{cpt } \text{tst } \psi c_1$
	$\text{mayT } c_2$	$\text{discr } c_2$	$\varphi c_2$	$\psi c_2$
While $\text{tst } d$	False	$\text{discr } d$	$\text{cpt } \text{tst } \varphi d$	False
Par $c_1 c_2$	$\text{mayT } c_1$	$\text{discr } c_1$	$\varphi c_1$	$\psi c_1$
	$\text{mayT } c_2$	$\text{discr } c_2$	$\varphi c_2$	$\psi c_2$

$$\varphi \in \{\text{siso}, \approx_s, \approx_{01T}, \approx_{wT}\} \quad \psi \in \{\approx_{01}, \approx_w\}$$

$$\psi_T \equiv \begin{cases} \approx_{01T}, & \text{if } \psi = \approx_{01} \\ \approx_{wT}, & \text{if } \psi = \approx_w \end{cases}$$

**Fig. 4.** Compositionality table

Example 1 already illustrates most of the above bisimilarities. Here are some further illustrations that also take Prop. 1 into account (using the Example 1 notations).

**Example 3.** (1)  $c_3$  is self 01-bisimilar, as any two discreet processes are 01-bisimilar. (2) However,  $c_3$  is not self 01T-bisimilar, as shown by the following reasoning: depending on  $h$ ,  $c_3$  can move to  $d \equiv h := 1$ ;  $h := 2$  or  $e \equiv h := 3$ ; but  $d$  and  $e$  are not 01T-bisimilar, as  $d$  is not able to 01T-match the immediate terminating step from  $e$ . (3) The above is not a problem for weak T-bisimilarity though, since here  $d$  can catch up with  $e$  by taking multiple steps. Thus,  $c_3$  is self weakly T-bisimilar (as any two discreet processes with finite behavior are weakly T-bisimilar).

- (4)  $c_5 \equiv c_3$ ;  $l := 4$  is self weakly T-bisimilar, since alternative executions (starting in indistinguishable states) of its first part  $c_3$  are able to  $\approx_{\text{WT}}$ -synchronize, so that they can proceed strongly synchronously with the remaining non-discreet step  $l := 4$ .
- (5) However,  $c_5$  is not self 01-bisimilar since the above  $e$ -continuation of  $c_3$  is able to terminate first, putting itself in a position to take the non-discreet step  $l := 4$ , not available at that time for the other continuation,  $d$ .
- (6) while  $h = 0$  do  $h := 0$  is discreet, hence self 01-bisimilar, but not weakly T-bisimilar, as a diverging execution from  $h = 0$  cannot match a terminating one from  $h \neq 0$ .

The weak and 01-bisimilarities provide the most fruitful notions in type-system approaches to noninterference. (The others—self isomorphism, discreteness and strong bisimilarity—are too harsh requirements, but, as we shall see, turn out as useful auxiliaries.) Thus, Smith and Volpano [29] focus on termination-sensitive weak bisimilarity. On the other hand, Boudol and Castellani [5, 6] prefer termination-insensitive 01-bisimilarity, while later Boudol [4] also considers weak bisimilarity, but in its termination-insensitive form. In these works, the newly introduced bisimilarities are not formally compared with preexisting ones—instead, the focus is on comparing the end-product type systems, i.e., the *rely* side of the contract (while the bisimilarities are the *guarantee* part). In order to properly revisit and compare type-system results, we first need an analysis of compositionality for these bisimilarities.

## 4 Compositionality

We now move to the central concept of this paper—compositionality of noninterference w.r.t. the language constructs.

An atom  $atm$  is called  $\sim$ -preserving, written  $\text{pres } atm$ , if  $\forall s. \text{aexec } atm \ s \sim s$ ; it is called  $\sim$ -compatible, written  $\text{cpt } atm$ , if  $\forall s \ t. s \sim t \implies \text{aexec } atm \ s \sim \text{aexec } atm \ t$ . A test  $tst$  is called  $\sim$ -compatible, written  $\text{cpt } tst$ , if  $\forall s \ t. s \sim t \implies \text{tval } tst \ s = \text{tval } tst \ t$ . In the setting of Example 2 for atoms,  $\sim$ -preservation means no assignment to low variables and  $\sim$ -compatibility means no direct leaks, i.e., no assignment to low variables of expressions depending on high variables (high expressions). Moreover, for tests,  $\sim$ -compatibility means no dependence on high variables.

**Prop 2.** The compositionality facts stated in Fig. 4 hold.

Here is how to read Fig. 4. The first column lists the possible forms of a command  $c$  ( $c$  may be an atom  $atm$ , or have the form  $\text{Seq } c_1 \ c_2$ , etc.). The next columns list conditions under which the predicates stated on the first row hold for  $c$ . Thus, e.g., row 3 column 3 says: if  $\text{discr } c_1$  and  $\text{discr } c_2$  then  $\text{discr } (\text{Seq } c_1 \ c_2)$ . The horizontal line in row 3 column 5 represents an “or” – thus, row 3 column 5 says: if either  $[\psi_{\top} \ c_1 \ \text{and } \psi \ c_2]$  or  $[\psi \ c_1 \ \text{and } \text{discr } c_2]$  then  $\psi \ (\text{Seq } c_1 \ c_2)$ . The involved bisimilarities are considered in their unary, “self” form, e.g.,  $\psi \ c$  means  $c \approx_{01} c$  or  $c \approx_w c$ .

**Example 4.** The informal arguments in Examples 1 and 3 can be made rigorous using the compositionality table in Fig. 4 in conjunction with the implication graph in Fig. 3. For instance,  $c_4$  from Example 1 has the form  $\text{Seq } (l := 4) \ c_3$ , where  $c_3$  has the form  $\text{if } (h = 0) \ (\text{Seq } (h := 1) \ (h := 2)) \ (h := 3)$ . According to the table, for  $c_4 \approx_{01} c_4$ , it suffices

that  $(l := 4) \approx_{\text{oit}} (l := 4)$  and  $c_3 \approx_{\text{o1}} c_3$ . The former is true by the table, since  $l := 4$  is compatible. However, the table cannot help (yet) in proving  $c_3 \approx_{\text{o1}} c_3$ , because there the required side condition is  $\text{cpt } (h = 0)$ , which does not hold. Therefore we turn to the implication graph, and try to prove the fact for one of the predecessors of  $\approx_{\text{o1}}$ . One predecessor is  $\approx_{\text{oit}}$ , which again requires  $\text{cpt } (h = 0)$ , and so does its predecessor  $\approx_s$ , and so does the predecessor of the latter,  $\text{siso}$ , which is a bottom node—therefore this path fails. The other predecessor of  $\approx_{\text{o1}}$  is  $\text{discr}$ , for which the table does not require the problematic side-condition. And the proof of  $\text{discr } c_3$  goes smoothly according to the table, since it is reduced to  $\text{discr } (\text{Seq } (h := 1) (h := 2))$  and  $\text{discr } (h := 3)$ , and further to  $\text{discr } (h := 1)$ ,  $\text{discr } (h := 2)$  and  $\text{discr } (h := 3)$ , all being true by  $\sim$ -preservation.

Note that we appeal to the Fig. 3 graph whenever the table result is not sufficiently strong, i.e., the given security notion is not sufficiently compositional w.r.t. the given language construct. For this table-and-graph proof technique, it is instructive to compare the termination-sensitive security notions with the termination-insensitive ones, that is,  $\varphi$  with  $\psi$  in Fig. 4.  $\varphi$  is more compositional than  $\psi$  w.r.t.  $\text{Seq}$ . (In fact, if interactivity is responsible for  $\text{Par}$ -compositional, termination-sensitiveness can be deemed responsible for  $\text{Seq}$ -compositional.) Indeed, for  $\psi$  ( $\text{Seq } c_1 c_2$ ) to go through, the table requires strengthening  $\psi$  either for  $c_1$  to its termination-sensitive variant,  $\psi_{\text{T}}$ , or for  $c_2$  to discreteness. A consequence of this is also the lack of compositionality of  $\psi$  w.r.t.  $\text{While}$  (since the semantics of  $\text{While}$  involves iteration of  $\text{Seq}$ ). On the other hand,  $\psi$  enjoys better compositionality w.r.t.  $\text{If}$ . This is not visible by looking at the table alone, where the  $\text{If}$  rules of  $\varphi$  and  $\psi$  are the same, and they are both conditioned by the  $\sim$ -compatibility of  $tst$ . The difference appears when  $tst$  is not compatible—then, according to the graph, unlike  $\varphi$ ,  $\psi$  can “fall back” on  $\text{discr}$ , which does not require  $tst$  to be compatible. Indeed, unlike  $\varphi$ ,  $\psi$  is above  $\text{discr}$  in the graph. Note that, among the  $\varphi$ 's,  $\approx_{\text{wt}}$  is the best located with this respect, since it is above the conjunction of  $\text{discr } c$  and  $\text{mayT } c$  in the graph. But this is still worse than  $\psi$ , since falling back on  $\text{discr } c \wedge \text{mayT } c$  forbids while loops, as shown in the table for  $\text{mayT}$ .

An interesting theoretical question is whether we can have the best of both worlds and define a relation that is both above discreteness in the graph and fully compositional w.r.t.  $\text{Seq}$ , without sacrificing compositionality with the other constructs. A positive answer to this question is presented next.

## 5 A More Compositional Security Notion

The rough idea of the proposed solution is as follows. If we knew that the whole program terminates, then discreteness would imply  $\approx_{\text{wt}}$ . And to integrate termination information into our coinductive interactivity, we note that, given a thread  $c$  running in parallel with others in a pool whose execution from a given state  $s$  is known to terminate, the following are true: (1) the execution of  $c$  alone starting in  $s$  must terminate; (2) between resumption points of the execution of  $c$ , the other threads are guaranteed to change the state in such a way that termination is preserved. This leads us to  $\approx_{\text{T}}$ , a relaxation of  $\approx_{\text{wt}}$  with interactivity restricted to  $\text{mustT}$  (“must terminate”) configurations, where  $\text{mustT}(c, s)$  is defined to mean that there exists no infinite chain  $(c_0, s_0), \dots, (c_n, s_n), \dots$  such that  $(c_0, s_0) = (c, s)$  and  $\forall i. (c_i, s_i) \rightarrow_c (c_{i+1}, s_{i+1})$ :

- $\text{match}_{\text{TMC}}^c \theta c d \equiv \forall s t c' s'. \text{mustT}(c, s) \wedge \text{mustT}(d, t) \wedge s \sim t \wedge (c, s) \rightarrow_c (c', s')$   
 $\implies (\exists d' t'. (d, t) \rightarrow_c^* (d', t') \wedge s' \sim t' \wedge \theta c' d')$
- $\text{match}_{\text{TMT}}^T c d \equiv \forall s t s'. \text{mustT}(c, s) \wedge \text{mustT}(d, t) \wedge s \sim t \wedge (c, s) \rightarrow_T s'$   
 $\implies (\exists t'. (d, t) \rightarrow_T^* t' \wedge s' \sim t')$
- $c \approx_T d \equiv \text{match}_{\text{TMC}}^c (\approx_T) c d \wedge \text{match}_{\text{TMT}}^T c d$

And, indeed,  $\approx_T$  achieves the targeted properties, as can be shown by an argument similar to those of Props. 1 and 2:

**Prop 3.** (1) The compositionality facts stated in Fig. 4 for  $\varphi$  also hold for  $\approx_T$ .  
 (2)  $\text{discr } c \implies c \approx_T c$ .

Note that  $\approx_T$  does not require, for the involved programs, termination (a liveness property), but rather preservation of termination (a safety property).  $\approx_T$  is weaker than  $\approx_{\text{WT}}$ , and neither weaker nor stronger than  $\approx_{\text{O1}}$  and  $\approx_{\text{W}}$ . The benefit of having  $\approx_T$  better suited than the other notions w.r.t. our table-and-graph reasoning is the availability of a more permissive syntactic criterion, as we detail next.

## 6 Syntactic Criteria

The (compositionality based) table-and-graph proof technique described in Example 4 can be automated, yielding a collection of recursive syntactic predicates corresponding to the various security notions. The recursive clauses for these predicates will simply perform the necessary lookups: first in the table, then, if needed, in the graph.

Before listing these clauses, we first simplify the Fig. 3 graph, noticing that  $\approx_s$  and  $\approx_{\text{O1T}}$  are redundant nodes on top of  $\text{siso}$ . Indeed, the compositionality conditions for  $\approx_s$  and  $\approx_{\text{O1T}}$  from the Fig. 4 table are identical to those of all nodes below, hence identical to those of  $\text{siso}$ . This means that, when proving  $c \approx_s c$  or  $c \approx_{\text{O1T}} c$ , one cannot do better than proving compositionality of the stronger (more desirable)  $\text{siso}$  notion of security. We therefore drop  $\approx_s$  and  $\approx_{\text{O1T}}$  from the graph. Fig. 5 shows this new graph, where  $\approx_T$  is also integrated. In the Fig. 4 table, we also redefine  $\psi_T$  by redirecting  $\approx_{\text{O1}}$  to  $\text{siso}$ :

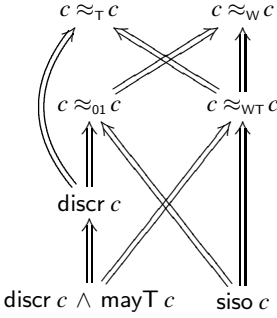
$$\psi_T \equiv \begin{cases} \text{siso}, & \text{if } \psi = \approx_{\text{O1}} \\ \approx_{\text{WT}}, & \text{if } \psi = \approx_{\text{W}}. \end{cases}$$

Let us introduce some notation for the Fig. 4 table and the Fig. 5 graph. A (syntactic) *constructor*  $Cns$  is any of the following:  $\text{Seq}$ ,  $\text{If } tst$  where  $tst \in \mathbf{test}$ ,  $\text{While } tst$  where  $tst \in \mathbf{test}$ ,  $\text{Par}$ . In addition, for uniformity, we also introduce a constructor  $\text{Atm } atm$  for every  $atm \in \mathbf{atom}$ , and assume  $\text{Atm } atm$  is the same as  $atm$ . Thus, any command  $c$  has the form  $Cns c_1 \dots c_k$ , where  $Cns$  is a constructor and  $c_1 \dots c_k$  are  $k$  commands, the *components* of  $c$ , with  $k$  either 0, 1 or 2, depending on  $Cns$  (it is 0 for  $\text{Atm } atm$ ).

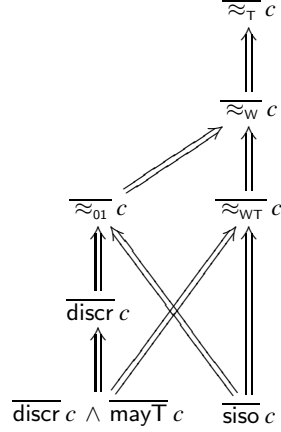
Henceforth, we let  $\chi$  range over the notions in the table, namely,  $\chi \in \{\text{mayT}, \text{discr}, \text{siso}, \approx_s, \approx_{\text{O1T}}, \approx_{\text{WT}}, \approx_{\text{O1}}, \approx_{\text{W}}, \approx_T\}$ . The table has an entry corresponding to every combination  $(\chi, Cns)$ , for which we define the following:

- $\text{side}_{\chi, Cns}$  is its *side condition*, i.e., the part of it not depending on the components. If this part is empty, we put  $\text{True}$ . E.g.,  $\text{side}_{\text{mayT}, \text{Atm } atm} = \text{side}_{\text{siso}, \text{Seq}} = \text{True}$ ,  $\text{side}_{\text{siso}, \text{If } tst} = \text{cpt } tst$ .





**Fig. 5.** Simplified implication graph of security notions



**Fig. 6.** Syntactic implications

- $\text{rcond}_{\chi, Cns}(c_1, \dots, c_k)$  is its *recursion condition*, i.e., the part involving the components of  $c$ . Again, if this part is empty, we put True. E.g.,  $\text{rcond}_{\text{mayT}, \text{Atm}} \text{atm} = \text{True}$ ,  $\text{rcond}_{\text{siso}, \text{Seq}}(c_1, c_2) = \text{rcond}_{\text{siso}, \text{If } \text{tst}}(c_1, c_2) = (\text{siso } c_1 \wedge \text{siso } c_2)$ .

For any element  $\chi$  in the graph, we let  $\text{Pred } \chi$  denote its set of predecessors. E.g.,  $\text{Pred } \text{siso} = \emptyset$ ,  $\text{Pred } \approx_{01} = \{\text{discr}, \text{siso}\}$ ,  $\text{Pred } \approx_W = \{\approx_{01}, \approx_{WT}\}$ .

Note that, for all  $\chi$ ,  $Cns$ , and  $c$  of the form  $Cns \ c_1 \dots c_k$ ,

- The table ensures that  $\text{side}_{\chi, Cns} \wedge \text{rcond}_{\chi, Cns}(c_1, \dots, c_k) \implies \chi \ c$ ;
- The graph ensures that  $(\bigvee_{\chi' \in \text{Pred } \chi} \chi' \ c) \implies \chi \ c$ .

We define, for each security notions  $\chi$ , a syntactic predicate  $\overline{\chi}$  on commands by turning the above implications into recursive clauses for each constructor  $Cns$ , where one first tries the table, and then, if the table fails, one tries the graph:

$$\overline{\chi} (Cns \ c_1 \dots c_k) \equiv \begin{cases} \overline{\text{rcond}_{\chi, Cns}}(c_1, \dots, c_k), & \text{if } \overline{\text{side}_{\chi, Cns}}(c_1, \dots, c_k), \\ \bigvee_{\chi' \in \text{Pred } \chi} \overline{\chi'} (Cns \ c_1 \dots c_k), & \text{otherwise,} \end{cases}$$

where  $\overline{\text{rcond}_{\chi, Cns}}$  and  $\overline{\text{side}_{\chi, Cns}}$  are  $\text{rcond}_{\chi, Cns}$  and  $\text{side}_{\chi, Cns}$  with all the involved security predicates  $\chi'$  replaced by their syntactic counterparts  $\overline{\chi'}$ .

For example, taking  $Cns = \text{If } \text{tst}$ , we have:

1.  $\overline{\text{discr}} (\text{If } \text{tst} \ c_1 \ c_2) = (\overline{\text{discr}} \ c_1 \wedge \overline{\text{discr}} \ c_2)$ .
  2.  $\overline{\text{siso}} (\text{If } \text{tst} \ c_1 \ c_2) = \begin{cases} \overline{\text{siso}} \ c_1 \wedge \overline{\text{siso}} \ c_2, & \text{if } \text{cpt } \text{tst} \\ \text{False}, & \text{otherwise.} \end{cases}$
  3.  $\overline{\approx_{01}} (\text{If } \text{tst} \ c_1 \ c_2) = \begin{cases} \overline{\approx_{01}} \ c_1 \wedge \overline{\approx_{01}} \ c_2, & \text{if } \text{cpt } \text{tst} \\ \overline{\text{discr}} (\text{If } \text{tst} \ c_1 \ c_2) \vee \overline{\text{siso}} (\text{If } \text{tst} \ c_1 \ c_2), & \text{otherwise} \end{cases}$
- = (by 1 and 2) =  $\begin{cases} \overline{\approx_{01}} \ c_1 \wedge \overline{\approx_{01}} \ c_2, & \text{if } \text{cpt } \text{tst} \\ \overline{\text{discr}} \ c_1 \wedge \overline{\text{discr}} \ c_2, & \text{otherwise.} \end{cases}$

(Recall that, when we instantiate  $\chi$  to a bisimilarity such as  $\approx_{\text{WT}}$ , we refer to its unary version, taking  $\chi c$  to be  $c \approx_{\text{WT}} c$ . Hence, an instance of  $\overline{\chi}$  is the unary predicate  $\overline{\approx_{\text{WT}}}$ .)

The proof of the following fact is now routine by structural induction on commands:

**Prop 4.** The syntactic criteria  $\overline{\chi}$  are sound for the security notions  $\chi$  in Fig. 5 in that  $\overline{\chi} c \implies \chi c$  for all commands  $c$ .

A remarkable property of the  $\overline{\chi}$ 's is that they preserve the Fig. 5 hierarchy of  $\chi$ 's. In fact, they actually refine it:

**Prop 5.** The implications listed in Fig. 6 hold.

The hierarchy refinement from Fig. 5 to Fig. 6 consists of the advance of  $\overline{\approx_{\tau}}$  to the top, even though  $\approx_{\tau}$  is not weaker than  $\approx_{\text{w}}$  or  $\approx_{\text{01}}$ . The reason why  $\overline{\approx_{\tau}}$  is weaker than  $\overline{\approx_{\text{w}}}$  is the following: The recursive definition of each  $\overline{\chi}$  is as *permissive* as is  $\chi$  *compositional*. And since  $\approx_{\tau}$  is at least as compositional as  $\approx_{\text{w}}$  and any other relation involved in its compositionality (here,  $\approx_{\text{WT}}$ ), the proof of  $\overline{\approx_{\text{w}}} c \implies \overline{\approx_{\tau}} c$  goes through by structural induction on  $c$ .

So far, our analysis was purely semantic and local: for semantic notions of security  $\chi$ , we studied compositionality w.r.t. each language construct, inferring from these syntactic criteria  $\overline{\chi}$  automatically. Now it is time to have a closer look at the recursive clauses of  $\overline{\chi}$  and see what they tell us about  $\overline{\chi}$  independently of  $\chi$ . First the easy cases:

- $\overline{\text{mayT}} c$  holds iff  $c$  does not contain while loops.
- $\overline{\text{discr}} c$  holds iff all atoms in  $c$  are  $\sim$ -preserving, a.k.a. high.
- $\overline{\text{siso}} c$  holds iff all tests in  $c$  are  $\sim$ -compatible, a.k.a. low, and all atoms are  $\sim$ -compatible.

$\overline{\text{siso}} c$  corresponds to a type system from Smith and Volpano [29] for scheduler independent security – this criterion is extremely harsh, forbidding high tests at If and While.

We now move to the more interesting cases.  $\overline{\approx_{\text{WT}}} c$  is equivalent to another, possibilistic type system from Smith and Volpano [29]. Here, high tests are allowed at If provided the branches are discreet, but are disallowed at While:  $\overline{\approx_{\text{WT}}} (\text{While } tst \ c) = \begin{cases} \overline{\approx_{\text{WT}}} c, & \text{if } \text{cpt } tst \\ \overline{\text{discr}} (\text{While } tst \ c) \wedge \overline{\text{mayT}} (\text{While } tst \ c), & \text{otherwise} \end{cases} = \begin{cases} \overline{\approx_{\text{WT}}} c, & \text{if } \text{cpt } tst \\ \text{False}, & \text{otherwise.} \end{cases}$

The above harsh condition on While is the starting point of work by Boudol and Castellani in [5, 6], where a type system equivalent to  $\overline{\approx_{\text{01}}}$  is introduced.  $\overline{\approx_{\text{01}}}$  allows high tests for While provided the body of the While is discreet. This is possible because, unlike  $\overline{\approx_{\text{WT}}}$ ,  $\overline{\approx_{\text{01}}}$  can fall back on  $\overline{\text{discr}}$ :

$$\overline{\approx_{\text{01}}} (\text{While } tst \ c) = \overline{\text{discr}} (\text{While } tst \ c) \vee \overline{\text{siso}} (\text{While } tst \ c) = \overline{\text{discr}} c \vee (\text{cpt } tst \wedge \overline{\text{siso}} c).$$

However, the price for this is a harsher clause for Seq (as we have seen, a limitation shared by all termination-insensitive notions). Indeed,  $\overline{\approx_{\text{WT}}}$  commutes smoothly with Seq as  $\overline{\approx_{\text{WT}}} (\text{Seq } c_1 \ c_2) = (\overline{\approx_{\text{WT}}} c_1 \wedge \overline{\approx_{\text{WT}}} c_2)$ , but  $\overline{\approx_{\text{01}}}$  needs either  $\overline{\text{siso}}$  on the left or  $\overline{\text{discr}}$  on the right:  $\overline{\approx_{\text{01}}} (\text{Seq } c_1 \ c_2) = (\overline{\text{siso}} c_1 \wedge \overline{\approx_{\text{01}}} c_2) \vee (\overline{\approx_{\text{01}}} c_1 \wedge \overline{\text{discr}} c_2)$ .

Thus,  $\overline{\approx_{\text{01}}}$  requires that either  $c_1$  has only low tests, or  $c_2$  has only high atoms. Hence, e.g., the command  $c_5$  from Example 1 is accepted by  $\overline{\approx_{\text{WT}}}$ , but rejected by  $\overline{\approx_{\text{01}}}$ .

An improvement of  $\overline{\approx_{\text{01}}}$  that accepts  $c_5$  also is proposed by Boudol in [4], where the idea is that, in the  $c_1$  part of Seq  $c_1 \ c_2$ , one should no longer restrict to low tests

everywhere, but rather only in places that may affect termination (i.e., inside While loops). Interestingly, this condition on  $c_1$  is the one imposed by  $\overline{\approx}_{\text{WT}}$ , and therefore the approach of [4] can be seen as a carefully designed combination of  $\overline{\approx}_{\text{WT}}$  and  $\overline{\approx}_{\text{OI}}$ . Remarkably, it turns out to be equivalent to  $\overline{\approx}_w$ , whose Seq clause is:  $\overline{\approx}_w(\text{Seq } c_1 c_2) = (\overline{\approx}_{\text{WT}} c_1 \wedge \overline{\approx}_w c_2) \vee (\overline{\approx}_w c_1 \wedge \overline{\text{discr}} c_2)$ .

In the above cited work, the soundness of the proposed type systems (results corresponding to Prop. 4) are given rather elaborate proofs, defining global bisimulation relations that involve multiple language constructs combined in ingenious and ad hoc ways. These proofs are often hard to understand and mechanize. Moreover, they are not exploiting the uniformities, commonalities and inter-dependencies of the various approaches. By contrast, our proof methodology is entirely local and uniform: we choose a language construct and a notion of security, and essentially do our best at proving (partial) compositionality. Then syntactic criteria follow automatically by our table-and-graph method. We were pleasantly surprised to find that this general method could capture such a variety of ad hoc results.

Finally, we discuss  $\overline{\approx}_\tau$ , which is our own novel type-system-like criterion for non-interference. It turns out to be a natural extension of the original Volpano-Smith-Irvine typing of sequential programs [30], using the same clauses for the sequential part together with  $\overline{\approx}_\tau(\text{Par } c_1 c_2) = (\overline{\approx}_\tau c_1 \wedge \overline{\approx}_\tau c_2)$ . The reason why such a natural type system is absent from the literature is probably that its associated semantic notion of security,  $\approx_\tau$ , was overlooked.

$\overline{\approx}_\tau$  accepts the commands  $c_7$  from Example 1 and  $d \equiv c_7 \parallel l := 5$ , while the most permissive criterion studied so far,  $\overline{\approx}_w$ , rejects them. However, as discussed, the security property that  $\overline{\approx}_\tau$  guarantees,  $\approx_\tau$ , is different from  $\approx_w$ , the main restriction of  $\approx_\tau$  being that it only makes sense under the termination assumption. Thus,  $\overline{\approx}_\tau$  provides a useful guarantee for  $c_7$  and  $d$  only in cases when the initial state  $s$  ensures termination, here, if it has  $h \geq 0$ . On the other hand,  $\overline{\approx}_w$  rejects  $d$  out of fear that its  $c_7$  component may not terminate, which would yield the pipelining of the  $c_7$  termination channel into a standard channel for  $d$ . Termination knowledge excludes such behavior, and this is where the new criterion  $\overline{\approx}_\tau$  is advantageous.

## 7 After-Execution Noninterference

The bisimilarity-based notions of security studied so far are rather complex, assuming an elaborate attacker model that interacts continuously with program execution—we call these *during-execution* noninterference. Often one is interested in a more tractable notion, as an input-to-output property, such as: a command is secure if, upon execution starting in indistinguishable states, the result states (after the command has finished executing) are again indistinguishable. We call such input-to-output properties *after-execution* noninterference.

So what are the after-execution guarantees of the various bisimilarities from §3? To answer this, we need some terminology. Given a configuration  $(c, s)$ :

- A *finite execution trace starting in  $(c, s)$*  (*finite  $(c, s)$ -trace* for short) is a finite sequence of the form  $(c_0, s_0), (c_1, s_1), \dots, (c_{n-1}, s_{n-1}), s_n$  (consisting of a number of configurations followed by a state) such that  $(c_0, s_0) = (c, s)$ ,  $(c_i, s_i) \rightarrow_c (c_{i+1}, s_{i+1})$

for all  $i < n - 1$ , and  $(c_{n-1}, s_{n-1}) \rightarrow_{\tau} s_n$ . Then  $n$  is said to be the *length* of the trace and  $s_n$  the *final state* of the trace.

- An *infinite execution trace starting in  $(c, s)$*  (infinite  $(c, s)$ -trace for short) is an infinite sequence of the form  $(c_0, s_0), (c_1, s_1), \dots$  (consisting of configurations only) such that  $(c_0, s_0) = (c, s)$  and  $\forall i. (c_i, s_i) \rightarrow_c (c_{i+1}, s_{i+1})$ .

Given a finite  $(c, s)$ -trace  $tr$ ,  $\text{length}(tr)$  denotes its length and  $\text{fstate}(tr)$  denotes its final state. Thus, finite  $(c, s)$ -traces represent the terminating computations starting in  $(c, s)$ , and infinite  $(c, s)$ -traces the divergent computations starting in  $(c, s)$ . Note that  $(c, s)$  “must terminate” (as defined in §5) iff there exist no infinite  $(c, s)$ -traces. It is not hard to prove the following about the termination-sensitive bisimilarities:

- Prop 6.** (1) If  $c \approx_s c$  and  $s \sim t$ , then, for every finite  $(c, s)$ -trace  $tr$ , there exists a finite  $(c, t)$ -trace  $tr'$  with  $\text{fstate}(tr') \sim \text{fstate}(tr)$  and  $\text{length}(tr') = \text{length}(tr)$ .  
 (2) If  $c \approx_{01T} c$  and  $s \sim t$ , then, for every finite  $(c, s)$ -trace  $tr$ , there exists a finite  $(c, t)$ -trace  $tr'$  with  $\text{fstate}(tr') \sim \text{fstate}(tr)$  and  $\text{length}(tr') \leq \text{length}(tr)$ .  
 (3) If  $c \approx_{wT} c$  and  $s \sim t$ , then, for every finite  $(c, s)$ -trace  $tr$ , there exists a finite  $(c, t)$ -trace  $tr'$  with  $\text{fstate}(tr') \sim \text{fstate}(tr)$ .

Thus, for self strongly bisimilar commands, terminating executions starting in indistinguishable states have, up to indistinguishability, the same outcomes, obtained in the same amount of time—this means both standard (low data) channels and timing channels are secure here. For self weakly T-bisimilar commands, again the outcomes are the same up to indistinguishability, but timing channels are no longer secured. As usual, 01T-bisimilarity lies in between—there is a time guarantee, but weaker than perfect synchronization.

Now, turning to the termination-insensitive notions, during-execution security faces the difficulty that here terminating executions need not be matched by terminating executions. However, we can still prove a termination-conditioned result:

- Prop 7.** If  $\forall s'. \text{mustT}(c, s')$ , then Prop. 6(3) holds with  $\approx_{01}$  or  $\approx_w$  substituted for  $\approx_{wT}$ .

Thus, in the termination-insensitive case, the after-execution distinction between 01- and weak bisimilarity vanishes. As for the after-execution guarantee of our termination-sensitive security notion  $\approx_{\tau}$  from §5 it is weaker than that of  $\approx_{wT}$  (Prop. 6(3)), but stronger than that of  $\approx_{01}$  and  $\approx_w$  (Prop. 7):

- Prop 8.** If  $\text{mustT}(c, s)$ , then Prop. 6(3) holds with  $\approx_{\tau}$  substituted for  $\approx_{wT}$ .

## 8 Conclusions and More Related Work

This paper was concerned with systematizing and comparing existing type-system based noninterference results from the literature. As a technical tool, we have introduced a compositionality “table-and-graph” technique able to capture such results in a uniform way. The study also suggested a novel, suitably compositional, notion, the termination-interactive bisimilarity  $\approx_{\tau}$ .

Our approach has important precursors in the literature. Thus, [25] makes a strong case for compositionality, and illustrates how it can be used to extend to concurrency a noninterference result [1] in the style of Volpano and Smith. However, [25] does not pursue this idea systematically or devise a general technique as we do in this paper. Moreover, our bisimilarity-based treatment employs insight from process algebra [16] in general and from process algebra approaches to noninterference [8] in particular. In system-based security, [9, 11, 15] provide general frameworks for trace-based system security, the last two having a special focus on compositionality and the first also incorporating probabilistic systems.

Themes missing from the compositionality framework discussed in this paper are probabilistic noninterference [13, 26–28], dynamic thread creation [13, 25, 31] and scheduler independence [6, 13, 25, 31], known to be particularly problematic w.r.t. noninterference. Incorporating some of these features in our compositional setting is a goal for future research.

Another exciting future direction is a framework for proving concurrent noninterference by a combination of automated and interactive methods along the lines of approaches going beyond type systems [2, 7, 12]. This would follow a rely-guarantee paradigm [10], with information about the environment made available to individual threads by suitably relaxing interactivity. A step towards this direction is made by our termination-interactive bisimilarity  $\approx_{\tau}$ , where such context information is termination, but could in principle be any liveness property.

**Acknowledgements.** We are grateful to Jasmin Blanchette for lots of suggestions that have significantly improved the presentation of this paper, to Benedict Nordhoff and Peter Lammich for noticing various technical typos, and to the anonymous reviewers for useful comments.

## References

1. Agat, J.: Transforming out timing leaks. In: POPL, pp. 40–53 (2000)
2. Barthe, G., D’Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: IEEE Computer Security Foundations Workshop, pp. 100–114 (2004)
3. Barthe, G., Nieto, L.P.: Formally verifying information flow type systems for concurrent and thread systems. In: FMSE, pp. 13–22 (2004)
4. Boudol, G.: On Typing Information Flow. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 366–380. Springer, Heidelberg (2005)
5. Boudol, G., Castellani, I.: Noninterference for Concurrent Programs. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 382–395. Springer, Heidelberg (2001)
6. Boudol, G., Castellani, I.: Noninterference for concurrent programs and thread systems. Theoretical Computer Science 281(1-2), 109–130 (2002)
7. Darvas, Á., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
8. Focardi, R., Gorrieri, R.: Classification of Security Properties (Part i: Information Flow). In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, p. 331. Springer, Heidelberg (2001)

9. Halpern, J.Y., O'Neill, K.R.: Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.* 12(1) (2008)
10. Jones, C.B.: Specification and design of (parallel) programs. In: *IFIP Congress 1983*, pp. 321–332 (1983)
11. Mantel, H.: On the composition of secure systems. In: *IEEE Symposium on Security and Privacy*, pp. 88–101 (2002)
12. Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: *CSF 2011, Cernay-la-Ville, France*, pp. 218–232 (2011)
13. Mantel, H., Sudbrock, H.: Flexible Scheduler-Independent Security. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) *ESORICS 2010. LNCS*, vol. 6345, pp. 116–133. Springer, Heidelberg (2010)
14. Mantel, H., Sudbrock, H., Krauß, T.: Combining Different Proof Techniques for Verifying Information Flow Security. In: Puebla, G. (ed.) *LOPSTR 2006. LNCS*, vol. 4407, pp. 94–110. Springer, Heidelberg (2007)
15. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions, pp. 79–93 (May 1994)
16. Milner, R.: *Communication and concurrency*. Prentice Hall (1989)
17. Plotkin, G.D.: A structural approach to operational semantics. *J. Log. Algebr. Program.* 60–61, 17–139 (2004)
18. Popescu, A., Hölzl, J.: Possibilistic noninterference formalized in Isabelle/HOL. *Archive for Formal Proofs* (2012), [http://afp.sourceforge.net/entries/Possibilistic\\_Noninterference.shtml](http://afp.sourceforge.net/entries/Possibilistic_Noninterference.shtml)
19. Russo, A., Hughes, J., Naumann, D.A., Sabelfeld, A.: Closing Internal Timing Channels by Transformation. In: Okada, M., Satoh, I. (eds.) *ASIAN 2006. LNCS*, vol. 4435, pp. 120–135. Springer, Heidelberg (2008)
20. Russo, A., Sabelfeld, A.: Security for Multithreaded Programs Under Cooperative Scheduling. In: Virbitskaite, I., Voronkov, A. (eds.) *PSI 2006. LNCS*, vol. 4378, pp. 474–480. Springer, Heidelberg (2007)
21. Sabelfeld, A.: The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) *PSI 2001. LNCS*, vol. 2244, pp. 225–239. Springer, Heidelberg (2001)
22. Sabelfeld, A.: Confidentiality for Multithreaded Programs via Bisimulation. In: Broy, M., Zamulin, A.V. (eds.) *PSI 2003. LNCS*, vol. 2890, pp. 260–274. Springer, Heidelberg (2004)
23. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21(1), 5–19 (2003)
24. 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)
25. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: *IEEE Computer Security Foundations Workshop*, pp. 200–214 (2000)
26. Smith, G.: A new type system for secure information flow. In: *IEEE Computer Security Foundations Workshop*, pp. 115–125 (2001)
27. Smith, G.: Probabilistic noninterference through weak probabilistic bisimulation. In: *IEEE Computer Security Foundations Workshop*, pp. 3–13 (2003)
28. Smith, G.: Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security* 14(6), 591–623 (2006)
29. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: *ACM Symposium on Principles of Programming Languages*, pp. 355–364 (1998)
30. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. *Journal of Computer Security* 4(2,3), 167–187 (1996)
31. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: *IEEE Computer Security Foundations Workshop*, pp. 29–43 (2003)

# Noninterference for Operating System Kernels<sup>\*,\*\*</sup>

Toby Murray<sup>1,2</sup>, Daniel Matichuk<sup>1</sup>, Matthew Brassil<sup>1</sup>,  
Peter Gammie<sup>1</sup>, and Gerwin Klein<sup>1,2</sup>

<sup>1</sup> NICTA, Sydney, Australia

<sup>2</sup> School of Computer Science and Engineering, UNSW, Sydney, Australia  
{firstname.lastname}@nicta.com.au

**Abstract.** While intransitive noninterference is a natural property for any secure OS kernel to enforce, proving that the implementation of any particular general-purpose kernel enforces this property is yet to be achieved. In this paper we take a significant step towards this vision by presenting a machine-checked formulation of intransitive noninterference for OS kernels, and its associated sound and complete unwinding conditions, as well as a scalable proof calculus over nondeterministic state monads for discharging these unwinding conditions across a kernel's implementation. Our ongoing experience applying this noninterference framework and proof calculus to the seL4 microkernel validates their utility and real-world applicability.

**Keywords:** Information flow, refinement, scheduling, state monads.

## 1 Introduction

A primary function of any operating system (OS) kernel is to enforce security properties and policies. The classical security property of *noninterference* [8] formalises the absence of unwanted information flows within a system, and is a natural goal for any secure OS to aim to enforce. Here, the system is divided into a number of *domains*, and the allowed information flows between domains specified by means of an information flow *policy*  $\rightsquigarrow$ , such that  $d \rightsquigarrow d'$  if information is allowed to flow from domain  $d$  to domain  $d'$ . So-called *intransitive noninterference* [10] generalises noninterference to the case in which the relation  $\rightsquigarrow$  is possibly intransitive.

---

\* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

\*\* This material is in part based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-10-1-4105. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

While intransitive noninterference is a natural property for any secure OS kernel to enforce, proving that the implementation of any particular general-purpose kernel enforces this property is yet to be achieved. In this paper we take a significant step towards this vision by presenting a machine-checked formulation of intransitive noninterference for OS kernels, and its associated sound and complete proof obligations (called *unwinding conditions*), as well as a scalable proof calculus over nondeterministic state monads for discharging these unwinding conditions across a kernel’s implementation. Both our noninterference formulation and proof calculus are termination-insensitive, under the assumption that a noninterference verification for an OS kernel is performed only after proving that its execution is always defined (and thus every system call always terminates). Our experience applying this noninterference framework and proof calculus to the seL4 microkernel [11] validates their utility and real-world applicability.

Our intransitive noninterference formulation improves on traditional formulations [10, 16, 19, 21] in two ways that make it more suitable for application to OS kernels. Firstly, traditional formulations of intransitive noninterference assume a static mapping  $\text{dom}$  from actions to domains, such that the domain  $\text{dom } a$  on whose behalf some action  $a$  is being performed can be determined solely from the action itself. No such mapping exists in the case of an OS kernel, which must infer this information at run-time. For instance, when a system call occurs, in order to work out which thread has requested the system call the kernel must consult the data-structures of the scheduler to determine which thread is currently running. This prevents traditional noninterference formulations from being able to reason about potential information flows that might occur via these scheduling data structures. An example would be a scheduler that does not properly isolate domains by basing its decision about whether to schedule a Low thread on whether a High thread is runnable. Our noninterference formulation makes  $\text{dom}$  dependent on the current state  $s$ , in order to overcome this problem, such that the domain associated with some action  $a$  that occurs from state  $s$  is  $\text{dom } a \ s$ . This makes the resulting noninterference formulation entirely state-dependent and complicates the proofs of soundness for our unwinding conditions. Proving that a system satisfies these unwinding conditions (and therefore our formulation of noninterference) requires showing that the scheduler does not leak information via its scheduling decisions.

Secondly, while phrased for (possibly) nondeterministic systems, our noninterference formulation is preserved by refinement. As explained later, this requires it to preclude all *domain-visible* nondeterminism, which necessarily abstracts away possible sources of information. Being preserved by refinement is important in allowing our noninterference formulation to be proved of real kernels at reasonable cost, as it can be proved about an abstract specification and then transported to the more complex implementation by refinement. In the case of seL4, this allows us to prove noninterference about a mostly-deterministic refinement [13] of its abstract *functional* specification, which its C implementation has been proved to refine [11], in order to conclude it of the implementation. Our experience to date suggests that reasoning about seL4’s functional specification requires an order of magnitude less effort than reasoning directly about the implementation [12].



Our proof calculus resembles prior language-based frameworks for proving termination-insensitive confidentiality (and other relational) properties of programs [1, 2, 4]; however, it is better suited than these frameworks for general-purpose OS kernels. Firstly, our calculus aims not at generality but rather at scalability, which is essential to enable its practical application to entire OS kernels. Secondly, it is explicitly designed for reasoning about systems for which no complete static assignment of memory locations or program variables to security domains exists. As is the case with a general-purpose OS kernel like seL4 that implements a dynamic access control system, whether a memory location is allowed to be read by the currently running thread depends on the access rights that the current thread has, if any, to that location. In a microkernel like seL4 that implements virtual memory, this of course depends on the current virtual memory mappings for the currently running thread. Thus, like the mapping of actions to domains, the mapping of memory locations to domains is also state-dependent in a general-purpose OS kernel. Our proof calculus is tuned to tracking and discharging these kinds of state-dependent proof obligations that arise when reasoning about confidentiality in such a system. These manifest themselves as preconditions on confidentiality statements about individual function calls that we discharge using a monadic Hoare logic and its associated VCG [6]. Our calculus is specially tuned so that this same VCG engine can automate its application, without modification, to automatically prove confidentiality statements for those functions that do not read confidential state (i.e. the vast majority of them), given appropriate user-supplied loop invariants.

Our experience applying this calculus to seL4 suggests that it scales very well to real-world systems. So far we have used it to prove confidentiality for 98% of the functions in the abstract seL4 specification in under 15 person-months. The remaining fraction comprises nondeterministic functions that abstract away from sources of confidential information — i.e. parts of the specification that are too abstract to allow correct reasoning about confidentiality. We are in the process of making these parts of the specification more concrete to produce a refinement of the functional specification, which seL4’s C implementation refines in turn, suitable for reasoning about confidentiality [13]. We have already done this and proved confidentiality for the *revoke* system call, which is the kernel’s most complex code path. The remaining functions are currently in progress.

In this paper, [Section 2](#) presents our noninterference formulation for OS kernels and its associated unwinding conditions. In [Section 3](#) we present our proof calculus for discharging these unwinding conditions across an entire kernel. [Section 4](#) considers related work before [Section 5](#) concludes. All theorems and definitions in this paper have been generated directly from the interactive theorem prover Isabelle/HOL [14] in which all of our work was carried out.

## 2 Noninterference

Our noninterference formulation for OS kernels extends von Oheimb’s notion of *noninfluence* [21]. We formalise noninterference over de Roever-Engelhardt

style *data types* [7], which can be thought of as automata with a supported theory of refinement, allowing us to prove that our noninterference formulation is preserved under refinement. We first introduce the data type formalism and the notion of refinement, before presenting our noninterference formulation and its associated unwinding conditions. We prove that the unwinding conditions are sound and complete for our noninterference formulation. We explain how our unwinding conditions (and, hence, our noninterference formulation) require us to prove the absence of information leaks through scheduling decisions. Lastly we show how our noninterference formulation is preserved by refinement.

## 2.1 Data Types and Refinement

We model an OS kernel as a state machine whose transitions include processing an interrupt or exception, performing a system call, and ordinary user-level actions like reading and writing user-accessible memory. We use the terms *event* and *action* interchangeably to refer to an automaton's individual transitions.

A data type automaton  $A$  is simply a triple comprising three functions: an initialisation function  $\text{Init}_A :: \text{state} \Rightarrow \text{istate set}$  that maps individual *observable* states to sets of corresponding internal states, an internal step relation  $\text{IStep}_A :: \text{event} \Rightarrow (\text{istate} \times \text{istate}) \text{ set}$ , and a final projection function  $\text{Fin}_A :: \text{istate} \Rightarrow \text{state}$  that maps individual internal states to corresponding individual observable states. For a data type  $A$  and initial observable state  $s :: \text{state}$  and sequence of transitions  $as$ , let  $\text{execution } A \ s \ as$  denote the set of observable states that can be reached by  $A$  performing  $as$ .  $\text{execution } A \ s \ as$  operates by first applying  $\text{Init}_A$  to the observable state  $s$  to produce a set of corresponding initial internal states. It then computes a set of resulting internal states by repeatedly applying  $\text{IStep}_A$  to each event  $a$  in  $as$  in turn to arrive at a set of final internal states. To each of these it applies  $\text{Fin}_A$  to obtain the set of final observable states.

$$\text{execution } A \ s \ as \equiv \text{Fin}_A \text{ ' foldl } (\lambda S \ a. \text{IStep}_A \ a \ \text{' } S) (\text{Init}_A \ s) \ as$$

Here  $R \ \text{' } S$  and  $f \ \text{' } S$  are the relational images of the set  $S$  under the relation  $R$  and function  $f$  respectively, and  $\text{foldl}$  is the standard fold function on lists.

A data type  $C$  refines data type  $A$ , written  $A \sqsubseteq C$ , when its behaviours are a subset of  $A$ 's.

$$A \sqsubseteq C \equiv \forall s \ as. \text{execution } C \ s \ as \subseteq \text{execution } A \ s \ as$$

## 2.2 System Model

Let  $A$  be an automaton, whose observable state is of type *state*, and  $s_0 :: \text{state}$  denote the initial observable state from which execution of  $A$  begins. Let *reachable*  $s$  denote that observable state  $s$  is reachable from  $s_0$ :

$$\text{reachable } s \equiv \exists as. s \in \text{execution } A \ s_0 \ as$$

As occurs in OS kernels generally, we assume that every event is always enabled.

$$\text{reachable } s \longrightarrow (\exists s'. s' \in \text{execution } A \ s \ as)$$

Let the function **Step** characterise the single-step behaviour of the system:

$$\text{Step } a \equiv \{(s, s') \mid s' \in \text{execution } A \ s \ [a]\}$$

For the information flow policy, we assume a set of security domains and a reflexive relation  $\rightsquigarrow$  that specifies the allowed information flows between domains, where  $d \rightsquigarrow d'$  implies that information is allowed to flow from domain  $d$  directly to  $d'$ . Noninterference asserts that no information flows outside of  $\rightsquigarrow$  can occur.

For each domain  $d$ , let  $\overset{d}{\sim}$  be an equivalence relation on observable states, such that  $s \overset{d}{\sim} t$  if and only if domain  $d$ 's state is identical in  $s$  and  $t$ . Here,  $d$ 's state will include the user-visible state that  $d$  can directly read, but might also include kernel-level state that the kernel might legitimately reveal to  $d$ . This relation is sometimes called an *unwinding relation*. When the system transitions directly from state  $s$  to state  $s'$  and  $s \overset{d}{\sim} s'$  for example, domain  $d$  has not been observably affected by this transition. For a set of domains  $D$ , let  $s \overset{D}{\approx} t \equiv \forall d \in D. s \overset{d}{\sim} t$ .

Traditional noninterference formulations associate a security domain  $\text{dom } e$  with each event  $e$  that occurs, which defines the domain that performed the event. Recall from [Section 1](#) that when a system call event occurs, the kernel must consult the data structures of the scheduler to determine which thread performed the system call, which will be the thread that is currently active. So events are not intrinsically associated with domains; rather, this association depends on part of the current state of the system which records the currently running domain.

Therefore, let  $\text{dom} :: \text{event} \Rightarrow \text{state} \Rightarrow \text{domain}$  be a function such that  $\text{dom } e \ s$  gives the security domain that is associated with event  $e$  in state  $s$ . When the scheduler's state is identical in states  $s$  and  $t$ , we expect that  $\text{dom } e \ s = \text{dom } e \ t$  for all events  $e$ . Formally, let  $\text{s-dom} :: \text{domain}$  be an arbitrary domain, whose state encompasses that part of the system state that determines which domain is currently active.  $\text{s-dom}$  stands for *scheduler domain*. Then we assume that for all events  $e$  and states  $s$  and  $t$

$$s \overset{\text{s-dom}}{\sim} t \longrightarrow \text{dom } e \ s = \text{dom } e \ t$$

Actions of the scheduling domain  $\text{s-dom}$  naturally include all those that schedule a new domain  $d$  to execute. We expect that when a domain  $d$  is scheduled, that  $d$  will be able to detect that it is now active, and so that an information flow might have occurred from  $\text{s-dom}$  to  $d$ . Since the scheduler can possibly schedule any domain, we expect that a wellformed information flow policy  $\rightsquigarrow$  will have an edge from  $\text{s-dom}$  to every domain  $d$ :

$$\text{s-dom} \rightsquigarrow d$$

In order to prevent  $\text{s-dom}$  from being a global transitive channel by which information can flow from any domain to any other, we require that information can never flow directly from any other domain  $d$  to  $\text{s-dom}$ , so

$$d \rightsquigarrow \text{s-dom} \longrightarrow d = \text{s-dom}.$$

This restriction forces us to prove that the scheduler’s decisions about which domain should execute next are independent of the other domains, which is typical scheduler behaviour in a separation kernel.

### 2.3 Formulating Noninterference

Traditionally [16], intransitive noninterference definitions make use of a `sources` function, whereby for a sequence of actions  $as$  and a domain  $d$ , `sources  $as$   $d$`  gives the set of domains that are allowed to pass information to  $d$  when  $as$  occurs. Because our `dom` function depends on the current state  $s$ , `sources` must do so as well. Therefore let `sources  $as$   $s$   $d$`  denote the set of domains that can pass information to  $d$  when  $as$  occurs, beginning in state  $s$ . The following definition is an extension of the standard one [16, 21] in line with our augmented `dom` function.

$$\begin{aligned} \text{sources } [] s d &= \{d\} \\ \text{sources } (a \cdot as) s d &= \end{aligned}$$

$$\begin{aligned} &\cup \{\text{sources } as s' d \mid (s, s') \in \text{Step } a\} \cup \\ &\{w \mid w = \text{dom } a s \wedge (\exists v s'. \text{dom } a s \rightsquigarrow v \wedge (s, s') \in \text{Step } a \wedge v \in \text{sources } as s' d)\} \end{aligned}$$

Here, we include in `sources  $(a \cdot as)$   $s$   $d$`  all domains that can pass information to  $d$  when  $as$  occurs from all successor-states  $s'$  of  $s$ , as well as the domain `dom  $a$   $s$`  performing  $a$ , whenever there exists some intermediate domain  $v$  that it is allowed to pass information to who in turn can pass information to  $d$  when the remaining events  $as$  occur from some successor state  $s'$  of  $s$ . An alternative, and seemingly more restrictive, definition would include only those domains that are present in all `sources  $as$   $s'$   $d$` , and include `dom  $a$   $s$`  only when some such  $v$  can be found for each `sources  $as$   $s'$   $d$` , where  $(s, s') \in \text{Step } a$ . However, as a consequence of Lemma 2 introduced later, this yields an equivalent noninterference formulation.

As is usual, the `sources` function is used to define a purge function, `ipurge`, in terms of which noninterference is formulated. Traditionally, for a domain  $d$  and action sequence  $as$ , `ipurge  $d$   $as$`  returns the sequence of actions  $as$  with all actions removed that are not allowed to (indirectly) influence  $d$  when  $as$  occurs [16]. Naturally, we must include the current state  $s$  in our `ipurge` function. However, for nondeterministic systems purging may proceed from a set  $ss$  of possible initial states. This leads to the following definition.

$$\begin{aligned} \text{ipurge } d [] ss &= [] \\ \text{ipurge } d (a \cdot as) ss &= \begin{cases} \text{if } \exists s \in ss. \text{dom } a s \in \text{sources } (a \cdot as) s d \\ \text{then } a \cdot \text{ipurge } d as (\bigcup_{s \in ss} \{s' \mid (s, s') \in \text{Step } a\}) \\ \text{else } \text{ipurge } d as ss \end{cases} \end{aligned}$$

Initially, the set  $ss$  will be a singleton containing one initial state  $s$ . Given a sequence of actions  $a \cdot as$  being performed from  $s$ , `ipurge` will keep the first action  $a$  if `dom  $a$   $s$   $\in$  sources  $(a \cdot as)$   $s$   $d$` , i.e. if this action is allowed to affect the target domain  $d$ . Purging then continues on the remaining actions  $as$  from the successor

states of  $s$  after  $a$ . On the other hand, if the action  $a$  being performed is not allowed to affect the target domain  $d$ , then it is removed from the sequence. For this reason, purging continues on the remaining actions  $as$  from the current state  $s$ , rather than its successors. We require the action to be able to affect the target domain in only one of the states  $s \in ss$  to avoid purging it. An alternative definition would instead place this requirement on all states  $s \in ss$ . Again however, because of [Lemma 2](#), this yields an equivalent noninterference formulation.

For states  $s$  and  $t$  and sequences of actions  $as$  and  $bs$  and domain  $d$ , let  $\text{uwr-equiv } s \text{ as } t \text{ bs } d$  denote when the contents of domain  $d$  is identical after executing  $as$  from  $s$  and  $bs$  from  $t$  in all resulting pairs of states. When  $\text{uwr-equiv } s \text{ as } t \text{ bs } d$  is true, domain  $d$  is unable to distinguish the cases in which  $as$  is executed from  $s$ , and  $bs$  is executed from  $t$ . Recall that we assume that every event is always enabled and that divergence never occurs on any individual execution step, under the assumption that noninterference is proved only after proving that a system's execution is always defined. This is why  $\text{uwr-equiv}$  and the following noninterference formulation are termination-insensitive.

$\text{uwr-equiv } s \text{ as } t \text{ bs } d \equiv$

$$\forall s' t'. s' \in \text{execution } A \text{ s as } \wedge t' \in \text{execution } A \text{ t bs} \longrightarrow s' \stackrel{d}{\sim} t'$$

Traditionally [\[16, 21\]](#) this property is defined using a projection function  $\text{out} :: \text{domain} \Rightarrow \text{state} \Rightarrow \text{output}$  so that, rather than testing whether  $s' \stackrel{d}{\sim} t'$  for final states  $s'$  and  $t'$ , it tests whether  $\text{out } d \text{ s}' = \text{out } d \text{ t}'$ . However, these traditional formulations invariably require the unwinding condition of *output consistency* which asserts that  $\text{out } d \text{ s}' = \text{out } d \text{ t}'$  whenever  $s' \stackrel{d}{\sim} t'$ , and construct the remaining unwinding conditions to establish precisely this latter relation. We avoid this indirection by discarding  $\text{out}$  entirely. One could re-phrase the noninterference formulation here in terms of  $\text{out}$  if necessary, in which case the addition of output consistency to the unwinding conditions presented here would be sufficient to prove the resulting noninterference property.

We now have the ingredients to express our noninterference formulation, which we derive as follows. Given two action sequences  $as$  and  $bs$ , a domain  $d$ , and an initial state  $s$  from which each sequence is executed, if  $\text{ipurge } d \text{ as } \{s\} = \text{ipurge } d \text{ bs } \{s\}$  then, when all events that are not allowed to affect  $d$  are removed from each sequence, they are both identical. So if none of these removed events can actually affect  $d$ , we should expect that  $d$  cannot distinguish the execution of one sequence from the other, i.e. that  $\text{uwr-equiv } s \text{ as } s \text{ bs } d$  should hold.

However, the only domains that should be able to affect  $d$  when  $as$  executes here are those in  $\text{sources } as \text{ s } d$ . So if  $s$  were modified to produce a state  $t$  from which  $bs$  was executed instead, we should expect  $\text{uwr-equiv } s \text{ as } t \text{ bs } d$  to hold so long as: (1)  $s$  and  $t$  agree on the state of all domains in  $\text{sources } as \text{ s } d$ , i.e.  $s \stackrel{\text{sources } as \text{ s } d}{\approx} t$ , and (2) the same domain is currently active in both, i.e.  $s \stackrel{s\text{-dom}}{\sim} t$ . This is our formulation of von Oheimb's *noninfluence* [\[21\]](#), denoted  $\text{noninfluence}$ .

noninfluence  $\equiv$

$\forall d \text{ as } bs \ s \ t.$

$$\begin{aligned} & \text{reachable } s \wedge \text{reachable } t \wedge s \stackrel{\text{sources } as \ s \ d}{\approx} t \wedge s \stackrel{s\text{-dom}}{\sim} t \wedge \\ & \text{ipurge } d \text{ as } \{s\} = \text{ipurge } d \text{ bs } \{s\} \longrightarrow \text{uwr-equiv } s \text{ as } t \text{ bs } d \end{aligned}$$

Note that, as a consequence of [Lemma 1](#) introduced later, replacing the term  $\text{ipurge } d \text{ bs } \{s\}$  by  $\text{ipurge } d \text{ bs } \{t\}$  here yields an equivalent property.

noninfluence might be too strong a property for systems with a pre-determined static schedule that is fixed for the entire lifetime of the system and known to all domains. If every domain always knows the exact sequence of events that must have gone before whenever it executes, then purging makes less sense. For these kinds of systems, an analogue of von Oheimb's weaker notion of *nonleakage* might be more appropriate. We denote this property *nonleakage*.

$$\text{nonleakage} \equiv \forall as \ s \ t \ d. \text{reachable } s \wedge \text{reachable } t \wedge s \stackrel{s\text{-dom}}{\sim} t \wedge s \stackrel{\text{sources } as \ s \ d}{\approx} t \longrightarrow \text{uwr-equiv } s \text{ as } t \text{ as } d$$

Naturally, noninfluence implies nonleakage.

## 2.4 Unwinding Conditions

A standard proof technique for noninterference properties involves proving so-called *unwinding conditions* [16](#) that examine individual execution steps of the system in question. We introduce two unwinding conditions. The first is sound and complete for nonleakage. The addition of the second to the first is sound and complete for noninfluence.

Both of these conditions examine individual execution steps of the system, and assert that they must all satisfy specific properties. As is usual with noninterference, we would like to conclude that these same properties are true across all runs of the system. However, this rests on the assumption that a run of the system, say in which it performs some sequence of actions  $as$ , is equivalent to performing a sequence of one-step executions for each of the events in  $as$  in turn.

This is formalised by the following function  $\text{Run}$ , which takes a step function  $\text{Stepf}$ , and repeatedly applies it to perform a sequence of actions  $as$  by executing each action in  $as$  in turn.

$$\begin{aligned} \text{Run } \text{Stepf } [] &= \{(s, s) \mid \text{True}\} \\ \text{Run } \text{Stepf } (a:as) &= \text{Stepf } a \circ \text{Run } \text{Stepf } as \end{aligned} \quad \text{Like ours, traditional unwinding con-}$$

ditions are predicated on the assumption that  $\text{reachable } s \longrightarrow \text{execution } A \ s \ as = \{s' \mid (s, s') \in \text{Run } \text{Step } as\}$  (assuming naturally that  $\text{reachable } s_0$  too). While this is valid for traditional noninterference formulations, in which their execution is defined exactly in this way [21](#), it is not always true for an arbitrary data-type automaton of the kind introduced in [Section 2.1](#) over which our noninterference properties are defined. However, for most well behaved data types this condition is true, and certainly holds for those that model the seL4 functional specification and its C implementation. Thus we restrict our attention to those automata  $A$  that satisfy this assumption and return to our unwinding conditions.

The first unwinding condition is a confidentiality property, while the second is an integrity property. The confidentiality property we denote *confidentiality-u*, and resembles the conjunction of von Oheimb’s *weak step consistency* and *step respect* [21] for deterministic systems; however, we require it to hold for *all* successor states and to take into account the scheduler domain *s-dom*.

$$\text{confidentiality-u} \equiv \forall a d s t s' t'. \text{reachable } s \wedge \text{reachable } t \wedge s \stackrel{\text{s-dom}}{\sim} t \wedge s \stackrel{d}{\sim} t \wedge (\text{dom } a s \rightsquigarrow d \longrightarrow s \stackrel{\text{dom } a s}{\sim} t) \wedge (s, s') \in \text{Step } a \wedge (t, t') \in \text{Step } a \longrightarrow s' \stackrel{d}{\sim} t'$$

This property says that the contents of a domain  $d$  after an action  $a$  occurs can depend only on  $d$ ’s contents before  $a$  occurred, as well as the contents of the domain  $\text{dom } a s$  performing  $a$  if that domain is allowed to send information to  $d$ . This condition alone allows  $d$  to perhaps infer that  $a$  has occurred, but not to learn anything about the contents of confidential domains.

The second unwinding condition is an integrity property, denoted *integrity-u*, and is essentially Rushby’s *local respect* [16] adapted to nondeterministic systems and again asserted for all successor states.

$$\text{integrity-u} \equiv \forall a d s s'. \text{reachable } s \wedge \text{dom } a s \not\rightsquigarrow d \wedge (s, s') \in \text{Step } a \longrightarrow s \stackrel{d}{\sim} s'$$

It says that an action  $a$  that occurs from some state  $s$  can affect only those domains that the domain performing the action,  $\text{dom } a s$ , is allowed to directly send information to. It prevents any domain  $d$  for which  $\text{dom } a s \not\rightsquigarrow d$  from even knowing that  $a$  has occurred.

The soundness proofs for these unwinding conditions are slightly more involved than traditional proofs of soundness for unwinding conditions. This is because our *sources* and *ipurge* functions are both state-dependent. The following lemma is useful for characterising those states that agree on *sources* and *ipurge*, under *confidentiality-u*, namely those related by  $\stackrel{\text{s-dom}}{\sim}$ .

**Lemma 1.**  $\text{confidentiality-u} \wedge \text{reachable } s \wedge \text{reachable } t \wedge s \stackrel{\text{s-dom}}{\sim} t \longrightarrow \text{sources } a s s d = \text{sources } a s t d \wedge \text{ipurge } d a s \{s\} = \text{ipurge } d a s \{t\}$

With this result, the proofs of the soundness of our unwinding conditions are similar to those for traditional non-state-dependent formulations of noninterference, since (as we explain shortly) *confidentiality-u* guarantees that the equivalence  $\stackrel{\text{s-dom}}{\sim}$ , asserted by *noninfluence* and *nonleakage*, is always maintained. The completeness proofs for these unwinding conditions are straightforward.

### Theorem 1 (Soundness and Completeness)

$$\text{nonleakage} = \text{confidentiality-u}, \text{ and } \text{noninfluence} = (\text{confidentiality-u} \wedge \text{integrity-u})$$

## 2.5 Scheduling

We said that our noninterference formulation requires us to show that the scheduler’s choices are independent of the other domains. To see why, consider when the domain  $d$  from our unwinding conditions is *s-dom*. Then *confidentiality-u* implies that *s-dom* can never be affected by the state of the other domains:

$$\forall a s t s' t'. \text{reachable } s \wedge \text{reachable } t \wedge s \stackrel{s\text{-dom}}{\sim} t \wedge (s, s') \in \text{Step } a \wedge (t, t') \in \text{Step } a \longrightarrow s' \stackrel{s\text{-dom}}{\sim} t'$$

Thus confidentiality-u implies that  $\stackrel{s\text{-dom}}{\sim}$  is always maintained.

When  $\text{dom } a s \neq s\text{-dom}$ ,  $\text{dom } a s \not\rightarrow s\text{-dom}$ . So integrity-u implies that the scheduler domain can never be affected by the actions of the other domains:

$$\forall a s s'. \text{reachable } s \wedge \text{dom } a s \neq s\text{-dom} \wedge (s, s') \in \text{Step } a \longrightarrow s \stackrel{s\text{-dom}}{\sim} s'$$

## 2.6 Refinement

We now show that noninfluence and nonleakage are preserved by refinement. This means we can prove them of an abstract specification  $A$  and conclude that they hold for all concrete implementations  $C$  that refine it (i.e. for which  $A \sqsubseteq C$ ).

### Theorem 2 (noninfluence and nonleakage are Refinement-Closed)

When  $A \sqsubseteq C$ ,  $\text{noninfluence}_A \longrightarrow \text{noninfluence}_C$ , and  $\text{nonleakage}_A \longrightarrow \text{nonleakage}_C$

*Proof.* We will prove that each unwinding condition is closed under refinement, which implies that their conjunction is as well. The result then follows from [Theorem 1](#). Let  $A$  and  $C$  be two automata, and write  $\text{Step}_A$ ,  $\text{sources}_A$  etc. for those respective functions applied to  $A$  and similarly for  $C$ . Then, when  $A \sqsubseteq C$ ,  $C$ 's executions are a subset of  $A$ 's, so  $\text{reachable}_C s \longrightarrow \text{reachable}_A s$  and  $\text{Step}_C a \subseteq \text{Step}_A a$ . It is straightforward to show then that  $\text{integrity-}u_A \longrightarrow \text{integrity-}u_C$  and  $\text{confidentiality-}u_A \longrightarrow \text{confidentiality-}u_C$ , as required.  $\square$

As mentioned earlier, a consequence of being preserved by refinement is that our unwinding conditions tolerate very little nondeterminism. Specifically, if the unwinding conditions hold, a system must have no *domain-visible* nondeterminism, which is nondeterminism that can be observed by any domain. This is because any such nondeterminism could abstract from a confidential source of information that is present in a refinement, and so implies the existence of insecure refinements. The following lemma states this restriction formally.

### Lemma 2 (No Visible Nondeterminism).

$$\text{confidentiality-}u \wedge \text{reachable } s \wedge (s, s') \in \text{Step } a \wedge (s, s'') \in \text{Step } a \longrightarrow s' \stackrel{d}{\sim} s''$$

## 3 A Proof Calculus for Confidentiality for State Monads

Having explained our noninterference formulation, and in particular its unwinding conditions, we now present a proof calculus for discharging these unwinding conditions across an OS kernel. We have successfully applied this calculus to the seL4 microkernel, as part of ongoing work to prove that it enforces our noninterference formulation.



Our proof calculus operates over nondeterministic state monads, the formalism that underpins the seL4 abstract functional specification. Specifically, the internal steps of the automaton that embodies the seL4 functional specification are formalised as computations of a nondeterministic state monad. The state type of this monad is simply the internal state of the automaton, which for the seL4 functional specification is also identical to its observable state. The unwinding condition `integrity-u` asserts that the state before a single execution step is related to each final state after the execution step. It is naturally phrased as a Hoare triple, and discharged using standard verification techniques. For seL4, we have used a monadic Hoare logic and its associated verification condition generator (VCG) [6] to discharge this condition [17]. This leaves just the property `confidentiality-u`. It is this property that our confidentiality proof calculus addresses.

### 3.1 Nondeterministic State Monad

To prove confidentiality for an entire kernel specification, we need to be able to decompose it across that specification to make verification tractable. It is this challenge that our proof calculus addresses for nondeterministic state monads.

The type for this nondeterministic state monad is

$$state \Rightarrow (\alpha \times state) \text{ set} \times bool$$

That is, it is a function that takes a state  $s$  as its sole argument, and returns a pair  $p$ . The first component `fst`  $p$  is a set of pairs  $(rv, s')$ , where  $rv$  is a return-value and  $s'$  is the result state. Each such pair  $(rv, s')$  represents a possible execution of the monad. The presence of more than one element in this set implies that the execution is nondeterministic. The second part `snd`  $p$  of the pair returned by the monad is a boolean flag, indicating whether at least one of the computations has failed. Since our confidentiality property is termination insensitive, this flag can be ignored for the purpose of this paper.

Our proof calculus for confidentiality properties over this state monad builds upon the simpler proof calculus for Hoare triples [6] mentioned above. In this calculus, a precondition  $P$  is a function of type  $state \Rightarrow bool$ , i.e. a function  $P$  such that, a given state  $s$  satisfies  $P$  if and only if  $P s$  is true. Since a monad  $f$  returns a set of return-value/result-state pairs, a postcondition  $Q$  is a function of type  $\alpha \Rightarrow state \Rightarrow bool$ .  $Q$  may be viewed as a function that given a return-value  $rv$  and corresponding result-state  $s'$ , tells whether they meet some criteria. Alternatively,  $Q$  may be viewed as a function that, given some return-value  $rv$ , yields a state-predicate  $Q rv$  that tests validity of the corresponding result-state  $s'$ . We write such Hoare triples as  $\{P\} f \{Q\}$ , and define their meaning as follows.

$$\{P\} f \{Q\} \equiv \forall s. P s \longrightarrow (\forall (rv, s') \in \text{fst} (f s). Q rv s')$$

The proof calculus for Hoare triples of our nondeterministic state monad includes a mechanical rule application engine that acts as a VCG for discharging Hoare triples [6]. Later we discuss how we can apply this same engine to act as a VCG for discharging our confidentiality properties.

### 3.2 Confidentiality over State Monads

Observe that the property, *confidentiality-u*, addressed by our confidentiality proof calculus considers two *pre-states*,  $s$  and  $t$ , for which some equivalences hold, and then asserts that for all *post-states*,  $s'$  and  $t'$ , another equivalence holds. We formalise this for our nondeterministic state monad, generalising over the pre- and post-state equivalences, as the property  $\text{ev } A \ B \ P \ f$ , pronounced *equivalence valid*. Here,  $A$  and  $B$  are pre-state and post-state equivalence relations respectively (often called just the *pre-equivalence* and *post-equivalence* respectively),  $f$  is the monadic computation being executed and  $P$  is a precondition that the pre-states  $s$  and  $t$  are assumed to satisfy.

$$\text{ev } A \ B \ P \ f \equiv \forall s \ t. P \ s \wedge P \ t \wedge A \ s \ t \longrightarrow (\forall (r_a, s') \in \text{fst} (f \ s). \forall (r_b, t') \in \text{fst} (f \ t). r_a = r_b \wedge B \ s' \ t')$$

Note that  $\text{ev } A \ B \ P \ f$  also asserts that the return values from both executions of  $f$  are equal. This requires that these return-values be derived only from those parts of the system state that are identical between the two executions (i.e. those parts that the pre-equivalence  $A$  guarantees are identical). The purpose of the precondition  $P$  is to allow extra conditions under which the pre-equivalence  $A$  guarantees that confidentiality is satisfied. For instance, if  $f$  is a function that reads a region of user memory, the precondition  $P$  might include a condition that ensures that this region is covered by the pre-equivalence  $A$ .

To decompose this property across a monadic specification, we need to define proof rules for the basic monad operators, *return* and  $\gg=$  (pronounced “bind”). *return*  $x$  is the state monad that leaves the state unchanged and simply returns the value  $x$ . This means that if  $A$  holds for the pre-states, then  $A$  will hold for the post-states as well. Also, *return*  $x$  always returns the same value ( $x$ ) when called. This gives us the following proof rule.

$$\frac{}{\text{ev } A \ A \ (\lambda\_. \text{True}) \ (\text{return } x)} \text{RETURN-EV}$$

Note that this rule restricts the post-equivalence to be the same as the pre-equivalence. As we explain shortly, this turns out not to be a problem in practice.

$f \gg= g$  is the composite computation that runs  $f$ , and then runs  $g$  on the result, and is used to sequence computations together. Specifically,  $f \gg= g$  runs the computation  $f$  to obtain a return value  $rv$  and result state  $s'$ , and then calls  $g \ rv$  to obtain a second computation that is run on the state  $s'$ . Because  $f$  might be nondeterministic,  $f \gg= g$  does this for all pairs  $(rv, s')$  that  $f$  emits, taking the distributed union over all results returned from each  $g \ rv \ s'$ .

Because we want to be able to decompose the proof of  $\text{ev}$  across a specification, we need a proof rule for  $f \gg= g$  that allows us to prove  $\text{ev}$  for  $f$  and  $g$  separately, and then combine the results to obtain a result overall. The following proof rule, *BIND-EV*, does exactly that.

$$\frac{\forall rv. \text{ev } B \ C \ (Q \ rv) \ (g \ rv) \quad \text{ev } A \ B \ P' \ f \quad \{P''\} f \ \{Q\}}{\text{ev } A \ C \ (P' \ \text{and } P'') \ (f \ \gg= \ g)} \text{BIND-EV}$$

Here,  $P'$  and  $P''$  is the conjunction of preconditions  $P'$  and  $P''$ , i.e.  $P'$  and  $P'' \equiv \lambda s. P' s \wedge P'' s$ . BIND-EV can be read as a recipe for finding a precondition  $?P$  such that  $\text{ev } A \ C \ ?P \ (f \gg= g)$  is true. First, for any return value  $rv$  that  $f$  might emit, find some state-equivalence  $B$  and a precondition  $Q \ rv$ , which may mention  $rv$ , such that  $g \ rv$  yields post-states that satisfy the post-equivalence  $C$ . Secondly, find a precondition  $P'$  such that executing  $f$  yields post-states that satisfy the just found state-equivalence  $B$ . Finally, find a precondition  $P''$ , such that for all return-values  $rv$  emitted from executing  $f$ , their corresponding result-states satisfy  $Q \ rv$ . The desired precondition  $?P$  is then  $P'$  and  $P''$ .

This rule works because if  $\text{ev}$  is true for  $f$ , we know that both executions of  $f$  yield the same return-value, say  $rv$ , which means that the two subsequent executions both run the same computation,  $g \ rv$ . In the rare case that  $\text{ev}$  cannot be proved for  $f$  (say because  $f$  returns a value  $rv$  derived from confidential state), a more sophisticated rule is required that we introduce later in [Section 3.4](#).

### 3.3 Automating Confidentiality Proofs

Note that when  $C = A$ , we may define a simpler variant of BIND-EV, called BIND-EV', in which  $B$  and  $C$  are both  $A$ .

$$\frac{\forall rv. \text{ev } A \ A \ (Q \ rv) \ (g \ rv) \quad \text{ev } A \ A \ P' \ f \quad \{P''\} f \ \{Q\}}{\text{ev } A \ A \ (P' \ \text{and } P'') \ (f \gg= g)} \text{ BIND-EV}'$$

To apply this rule, we need only compute sufficient preconditions  $Q \ rv$ ,  $P'$  and  $P''$  for the relevant obligations. Our ordinary Hoare logic VCG can be applied to compute  $P''$ , of course, while BIND-EV' is itself a recipe for computing sufficient  $Q \ rv$  and  $P'$ . In other words, we may recursively apply BIND-EV' to compute appropriate  $Q \ rv$  and  $P'$ , given appropriate  $\text{ev}$  rules for the primitive monadic functions.

This is precisely the technique that we have taken to prove these statements across the seL4 functional specification. Specifically, at the top-level, the pre-equivalence of confidentiality-u, asserted for  $s$  and  $t$ , implies the post-equivalence,  $\overset{d}{\sim}$ , asserted for all  $s'$  and  $t'$ , because the pre-equivalence includes  $\overset{d}{\sim}$ . Hence, if we prove that the pre-equivalence is preserved, we can deduce that the post-equivalence must hold after each kernel event. This allows us to reason about a more restricted version of  $\text{ev}$  in which the pre- and post-equivalences are always identical, using rules like RETURN-EV and BIND-EV' above.

The rule-application engine developed previously [\[6\]](#) that acts as a VCG for Hoare triples over our nondeterministic state monad, can be applied directly without any modification to discharge  $\text{ev}$  statements by feeding it the appropriate rules. It requires rules like BIND-EV', to decompose these proofs into smaller goals, as well as appropriate rules, like RETURN-EV, to discharge the goals at the leaves of the proof tree. Familiar rules from prior work on proof methods for relational properties of programs [\[1, 2, 4\]](#) may be derived for other monadic functions, such as the one below for monadic while-loops. It establishes confidentiality for the loop under the invariant  $P$  when the loop body  $B$  maintains

confidentiality and the pre-equivalence  $A$  guarantees that both executions terminate together. The loop body  $B$  and condition  $C$  are both parametrised by a loop parameter  $n$ , which for subsequent loop iterations is the return-value of the previous iteration.

$$\frac{\forall s t n. A s t \wedge P n s \wedge P n t \longrightarrow C n s = C n t \quad \forall n. \{\!\{P n \text{ and } C n\}\!\} B n \{\!\{P\}\!\} \quad \forall n. \text{ev } A A (P n \text{ and } C n) (B n)}{\text{ev } A A (P n) (\text{whileLoop } C B n)} \text{WHILE-EV}$$

### 3.4 Proving the Functions That Read Confidential State

The approach so far allows very automatic proofs for functions that do not read any confidential state, and so always yield identical return-values  $rv$ . Because these functions make up the bulk of seL4, this is what our calculus has been tuned for. However, it is less well suited to functions that operate on confidential state without revealing it to unauthorised domains. Our approach requires confidentiality proofs for these kinds of functions to be performed more manually.

An example is the seL4 function `send-async-ipc`, which sends a message on an *asynchronous endpoint*. Asynchronous endpoints facilitate unidirectional communication, which implies that the act of sending on an asynchronous endpoint should not leak any information back to the sender. Sending such a message does require the kernel to read state outside of the sending domain (such as state in the endpoint); however, it should not reveal any of this state to the sender.

There is no guarantee, then, that when the two executions of `send-async-ipc` that `ev` compares each read the internal state of the endpoint in question, they will get the same result. This means their subsequent executions might behave differently to each other. Proving that `ev` holds in this case requires comparing two different executions, and showing that they establish the post-equivalence. This suggests that we should reason about a more general property than `ev` that can talk about two different executions, and allows return-values to differ.

These insights lead to the following property, called `ev2`.

$$\text{ev2 } A B R P P' f f' \equiv \forall s t. P s \wedge P' t \wedge A s t \longrightarrow (\forall (r_a, s') \in \text{fst } (f s). \forall (r_b, t') \in \text{fst } (f' t). R r_a r_b \wedge B s' t')$$

`ev2` takes two computations,  $f$  and  $f'$ , and two associated preconditions,  $P$  and  $P'$ . It also takes a return-value relation  $R$ , that it asserts holds for the return-values of  $f$  and  $f'$ . `ev2` generalises `ev`, specifically  $\text{ev } A B P f \equiv \text{ev2 } A B \text{op} = P P f f$ , where `op =` is the equality operator. One usually applies this equivalence to rewrite `ev` goals that cannot be proved by the VCG, into `ev2` goals. One then manually applies proof rules like in [Figure 1](#) to discharge these goals.

Applying `BIND-EV2` usually requires the human to come up with an appropriate intermediate return-value relation  $R'$  that will hold for the return values emitted from  $f$  and  $f'$ . As with `ev`, we usually work with a simpler rule in which  $(B = C) = A$ , which we omit for brevity. We suspect that techniques could be

$$\frac{\forall s t. P s \wedge P' t \wedge A s t \longrightarrow R x y}{\text{ev2 } A A R P P' (\text{return } x) (\text{return } y)} \text{RETURN-EV2}$$

$$\frac{\forall rv rv'. R' rv rv' \longrightarrow \text{ev2 } B C R (Q rv) (Q' rv') (g rv) (g' rv')}{\text{ev2 } A B R' P P' f f' \quad \{\!\{S\}\!\} f \{\!\{Q\}\!\} \quad \{\!\{S'\}\!\} f' \{\!\{Q'\}\!\}} \text{BIND-EV2}$$

$$\frac{}{\text{ev2 } A C R (P \text{ and } S) (P' \text{ and } S') (f \gg= g) (f' \gg= g')}$$

Fig. 1. VCG rules for ev2

borrowed from other work on automatically proving confidentiality properties of programs [2,18] to help automatically infer appropriate  $R'$ . However, because ev2 proofs are seldom required for seL4, we have not needed to implement them.

## 4 Related Work

Recently, Barthe et al. [3] presented a formalisation of isolation for an *idealised* model of a hypervisor, and its unwinding conditions. Like ours, their definition is based on von Oheimb's noninfluence [21]. As in traditional formalisations of noninterference, in their formulation actions are intrinsically linked to domains, and so it cannot reason about information leaks through scheduling decisions.

INTEGRITY-178B is a real-time operating system for which an isolation proof has been completed [15]. The isolation property proved is based on the GWVr2 information flow property [9], which bears similarities to the unwinding conditions for noninterference. Like ours, it is general enough to handle systems in which previous execution steps affect which is the entity that executes next. Unlike ours, it is defined only for deterministic systems. The exact relationship between GWVr2 and our conditions deserves further study.

Our formulation of information flow security is descendant from traditional *ipurge*-based formulations of intransitive noninterference (starting with Haigh and Young's [10]). Van der Meyden [19] argues that *ipurge*-based formulations of intransitive noninterference are too weak for certain intransitive policies, and proposes a number of stronger definitions. He shows that Rushby's unwinding conditions [16] are sufficient for some of these alternatives. Given the similarity of our unwinding conditions to Rushby's, we wonder whether our existing unwinding conditions may be sufficient to prove analogues of van der Meyden's definitions.

Others have presented noninterference conditions for systems with scheduling components. One recent example is van der Meyden and Zhang [20], who consider systems that run in lock-step with a scheduling component that controls which domain's actions are currently enabled. Their security condition for the scheduler requires that the actions of the High domain cannot affect scheduling decisions. Our formulation, in contrast, has the scheduler update a component of the system state that determines the currently running domain. This allows our scheduler security condition to require that scheduling decisions be unaffected not only by domain actions, but also by domain state.

A range of proof calculi and verification procedures for confidentiality properties, and other *relational* properties, have also been developed [1,2,4,5,18]. Unlike many of these, ours aims not at generality but rather at scalability. The simplicity of our calculus has enabled it to scale to the entire functional specification of the seL4 microkernel, whose size is around 2,500 lines of Isabelle/HOL, and whose implementation that refines this specification is around 8,500 lines of C.

## 5 Conclusion

We have presented a definition of noninterference for operating system kernels, with sound and complete unwinding conditions. We have shown how these latter can be implemented in a proof calculus for nondeterministic state monads with automation support. Our success in applying both of these to the seL4 microkernel, in an ongoing effort to prove that it enforces noninterference, attest to their practical utility and applicability to programs on the order of 10,000 lines of C.

**Acknowledgements.** We thank Kai Engelhardt, Sean Seefried, and Timothy Bourke for their comments on earlier drafts of this paper.

## References

1. Amtoft, T., Banerjee, A.: Information Flow Analysis in Logical Form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004)
2. Amtoft, T., Banerjee, A.: Verification condition generation for conditional information flow. In: FMSE 2007, pp. 2–11. ACM (2007)
3. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Formally Verifying Isolation and Availability in an Idealized Model of Virtualization. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 231–245. Springer, Heidelberg (2011)
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL 2004, pp. 14–25. ACM (2004)
5. Beringer, L.: Relational Decomposition. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 39–54. Springer, Heidelberg (2011)
6. Cock, D., Klein, G., Sewell, T.: Secure Microkernels, State Monads and Scalable Refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)
7. de Roeper, W.-P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press (1998)
8. Goguen, J., Meseguer, J.: Security policies and security models. In: IEEE Symp. Security & Privacy, Oakland, California, USA, pp. 11–20. IEEE (April 1982)
9. Greve, D.A.: Information security modeling and analysis. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 249–300. Springer (2010)
10. Haigh, J.T., Young, W.D.: Extending the noninterference version of MLS for SAT. Trans. Softw. Engin. 13, 141–150 (1987)

11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSOP, pp. 207–220. ACM (2009)
12. Klein, G., Murray, T., Gammie, P., Sewell, T., Winwood, S.: Provable security: How feasible is it? In: 13th HotOS, Napa, CA, USA, pp. 28–32. USENIX (May 2011)
13. Matichuk, D., Murray, T.: Extensible Specifications for Automatic Re-use of Specifications and Proofs. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 333–341. Springer, Heidelberg (2012)
14. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
15. Richards, R.J.: Modeling and security analysis of a commercial real-time operating system kernel. In: Hardin, D.S. (ed.) Design and Verification of Microprocessor Systems for High-Assurance Applications, pp. 301–322. Springer (2010)
16. Rushby, J.: Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International (December 1992)
17. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., Klein, G.: seL4 Enforces Integrity. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 325–340. Springer, Heidelberg (2011)
18. Terauchi, T., Aiken, A.: Secure Information Flow as a Safety Problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
19. van der Meyden, R.: What, Indeed, Is Intransitive Noninterference? In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 235–250. Springer, Heidelberg (2007)
20. van der Meyden, R., Zhang, C.: Information flow in systems with schedulers. In: 21st CSF, pp. 301–312. IEEE (June 2008)
21. von Oheimb, D.: Information Flow Control Revisited: Noninfluence = Noninterference + Nonleakage. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (eds.) ESORICS 2004. LNCS, vol. 3193, pp. 225–243. Springer, Heidelberg (2004)

# Compositional Verification of a Baby Virtual Memory Manager

Alexander Vaynberg and Zhong Shao

Yale University

**Abstract.** A virtual memory manager (VMM) is a part of an operating system that provides the rest of the kernel with an abstract model of memory. Although small in size, it involves complicated and interdependent invariants that make monolithic verification of the VMM and the kernel running on top of it difficult. In this paper, we make the observation that a VMM is constructed in layers: physical page allocation, page table drivers, address space API, etc., each layer providing an abstraction that the next layer utilizes. We use this layering to simplify the verification of individual modules of VMM and then to link them together by composing a series of small refinements. The compositional verification also supports function calls from less abstract layers into more abstract ones, allowing us to simplify the verification of initialization functions as well. To facilitate such compositional verification, we develop a framework that assists in creation of verification systems for each layer and refinements between the layers. Using this framework, we have produced a certification of BabyVMM, a small VMM designed for simplified hardware. The same proof also shows that a certified kernel using BabyVMM's virtual memory abstraction can be refined following a similar sequence of refinements, and can then be safely linked with BabyVMM. Both the verification framework and the entire certification of BabyVMM have been mechanized in the Coq Proof Assistant.

## 1 Introduction

Software systems are complex feats of engineering. What makes them possible is the ability to isolate and abstract modules of the system. In this paper, we consider an operating system kernel that uses virtual memory. The majority of the kernel makes an assumption that the memory is a large space with virtual addresses and a specific interface that allows the kernel to request access to any particular page in this large space. In reality, this entire model of memory is in the imagination of the programmer, supported by a relatively small but important portion of the kernel called the virtual memory manager. The job of the virtual memory manager is to handle all the complexities of the real machine architecture to provide the primitives that the rest of the kernel can use. This is exactly how the programmer would reason about this software system.

However, when we consider verification of such code, current approaches are mostly monolithic in nature. Abstraction is generally limited to abstract data types, but such abstraction can not capture changes in the semantics of computation. For example, it is impossible to use abstract data types to make virtual memory appear to work like physical memory without changing operational semantics. To create such abstraction, a



change of computational model is required. In the Verisoft project [11,18], the abstract virtual memory is defined by creating the CVM model from VAMP architecture. In AIM [7], multiple machines are used to define interrupts in the presence of a scheduler.

These transitions to more abstract models of computation tend to be quite rare, and when present tend to be complex. The previously mentioned VAMP-CVM jump in Verisoft abstracts most of kernel functionality in one step. In our opinion, it would be better to have more abstract computation models, with smaller jumps in abstraction. First, it is easier to verify code in the most abstract computational model possible. Second, smaller abstractions tend to be easier to prove and to maintain, while larger abstractions can be still achieved by composing the smaller ones. Third, more abstractions means more modularity; changes in the internals of one module will not have global effects.

However, we do not commonly see Hoare-logic verification that encourages multiple models. The likely reason is that creating abstract models and linking across them is seen as ad-hoc and tedious additional work. In this paper we show how to reduce the effort required to define models and linking, so that code verification using multiple abstractions becomes an effective approach. More precisely, our paper makes the following contributions:

- We present a framework for quickly defining multiple abstract computational models and their verification systems.
- We show how our framework can be used to define safe cross-abstraction linking.
- We show how to modularize a virtual memory manager and define abstract computational models for each layer of VMM.
- We show a complete verification of a small proof-of-concept virtual memory manager using the Coq Proof Assistant.

The rest of this paper is organized as follows. In Section 2, we give an informal overview of our work. In Section 3, we discuss the formal details of our verification and refinement framework. In Section 4, we specialize the framework for a simple C-like language. In Section 5, we certify BabyVMM, our small virtual memory manager. Section 6 discusses the Coq proof, and Section 7 presents related work and concludes.

## 2 Overview and Plan for Certification

We begin the overview by explaining the design of BabyVMM, our small virtual memory manager. First, consider the model of memory present in simplified hardware (left side of Figure 1). The memory is a storage system, which contains cells that can be read from or written to by the software. These cells are indexed by addresses. However, to facilitate indirection, the hardware includes a system called address translation (AT), which, when enabled, will cause all requests for specific addresses from the software to be translated. The AT system adds special registers to the memory system - one to enable or disable AT, and the other to point where the software-managed AT tables are located in memory. The fact that these tables are stored in memory is one of the sources of complexity in the AT system - updating AT tables requires updating in-memory tables, a process which goes through AT as well.

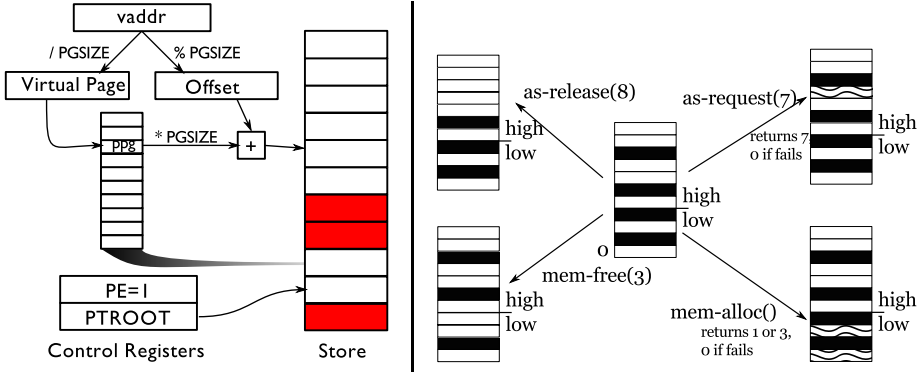


Fig. 1. Hardware (HW) and Address Space (AS) Models of Memory

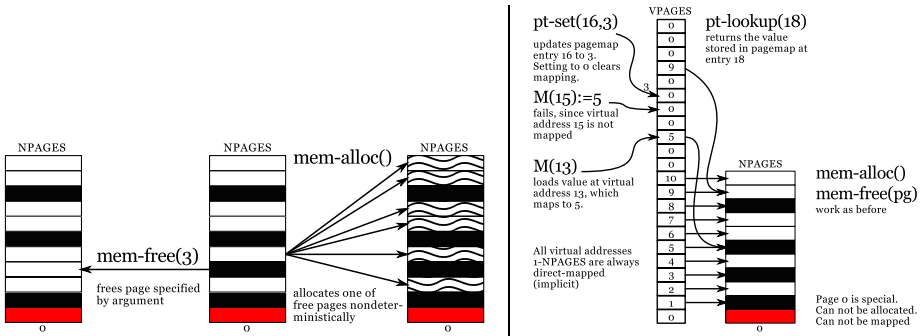


Fig. 2. Allocated (ALE) and Page Map (PMAP) Models of Memory

Because AT is such a complicated, machine-dependent, and general mechanism, BabyVMM creates an abstraction that defines specific restrictions on how AT will be used, and presents a simpler view of AT to the kernel. Although the abstract models of memory may differ depending on the features that the kernel may require, BabyVMM defines a very basic model, to which we refer as the address space (AS) model of memory (right side of Figure 1). The AS model replaces the small physical memory with a larger virtual address space with allocatable pages and no address translation. The space is divided into high and low areas, where the low area is actually a window into physical memory (a pattern common in many kernels). Because of this distinction, the memory model has two sets of allocation functions, one for the “high” memory area where the programmer requests a specific page for allocation, and one for the “low” memory area, where the programmer can not pick which page to allocate.

However, creating an abstraction that makes the jump from the HW model directly to AS model is complex. As a result, we create two more intermediate models, which slowly build up the abstraction. The first model is ALE (left side of Figure 2), which incorporates allocation information into the hardware memory, requiring that programs

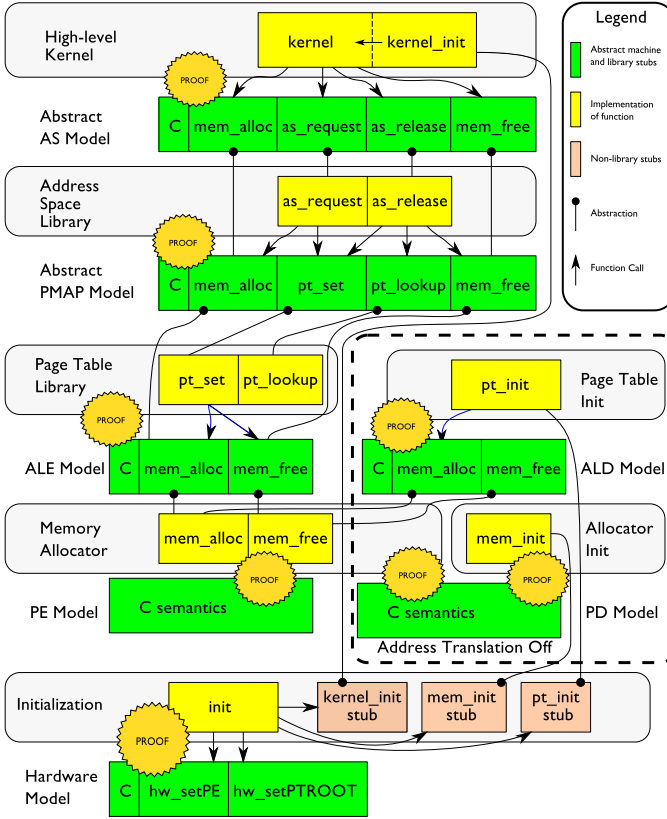


Fig. 3. Complete Plan for VMM Certification

only access memory locations that are marked allocated. The model adds primitives in the form of `mem_alloc` and `mem_free`, with semantics same as the ones in the AS model. Although this is not shown on the diagram, the ALE model still maintains the hardware’s AT mechanism.

The second intermediate level, which we call PMAP (right side of Figure 2) is designed to replace the hardware’s AT mechanism with an abstract one. The model features a page map that exists outside the normal memory space, unlike the lower level models. The page map maps virtual page numbers to physical page numbers, with a 0 value meaning invalid. In our particular design, the pagemap is always identity for the lower addresses, creating a window into physical memory from within the virtual space. The model still contains allocation primitives, and adds two more primitives, `pt_set` and `pt_lookup`, which update and lookup values in the pagemap.

Using these abstract memory models, we can construct the BabyVMM verification plan (Figure 3). The light-yellow boxes in the kernel represent the actual functions

<p>(State) <math>\mathbb{S} \in \Sigma</math></p> <p>(Operation) <math>\iota \in \Delta</math></p> <p>(Cond) <math>b \in \beta</math></p> <p>(CondInterp) <math>\Upsilon \in \beta \rightarrow \Sigma \rightarrow Prop</math></p>	<p>(State Predicate) <math>p \in \Sigma \rightarrow Prop</math></p> <p>(State Relation) <math>g \in \Sigma \rightarrow \Sigma \rightarrow Prop</math></p> <p>(Operational Semantics) <math>OS \in \{\iota \rightsquigarrow (p, g)\}^*</math></p> <p>(Language / Machine) <math>\mathcal{M} \in (\Sigma, \Delta, \beta, \Upsilon, OS)</math></p>
<p>where <math>\mathcal{M}(\iota) \triangleq \mathcal{M}.OS(\iota)</math> and <math>\mathcal{M}(b) \triangleq \mathcal{M}.\Upsilon(b)</math></p>	

**Fig. 4.** Abstract State Machine

<i>id</i>	$\triangleq (\lambda \mathbb{S}. True, \lambda \mathbb{S}. \lambda \mathbb{S}'. \mathbb{S}' = \mathbb{S})$
<i>fail</i>	$\triangleq (\lambda \mathbb{S}. False, \lambda \mathbb{S}. \lambda \mathbb{S}'. False)$
<i>loop</i>	$\triangleq (\lambda \mathbb{S}. True, \lambda \mathbb{S}. \lambda \mathbb{S}'. False)$
$(p, g) \circ (p', g')$	$\triangleq (\lambda \mathbb{S}. p \mathbb{S} \wedge \forall \mathbb{S}'. g \mathbb{S} \mathbb{S}' \rightarrow p' \mathbb{S}', \lambda \mathbb{S}. \lambda \mathbb{S}'. \exists \mathbb{S}'' . g \mathbb{S} \mathbb{S}' \wedge g' \mathbb{S}' \mathbb{S}'')$
$(p, g) \oplus_c (p', g')$	$\triangleq (\lambda \mathbb{S}. (p \mathbb{S} \wedge c \mathbb{S}) \vee (p' \mathbb{S} \wedge \neg c \mathbb{S}), \lambda \mathbb{S}. \lambda \mathbb{S}'. (c \mathbb{S} \wedge g \mathbb{S} \mathbb{S}') \vee (\neg c \mathbb{S} \wedge g' \mathbb{S} \mathbb{S}'))$
$(p, g) \supseteq (p', g')$	$\triangleq \forall \mathbb{S}. p \mathbb{S} \rightarrow p' \mathbb{S} \wedge \forall \mathbb{S}. \mathbb{S}'. g' \mathbb{S} \mathbb{S}' \rightarrow g \mathbb{S} \mathbb{S}'$

**Fig. 5.** Combinators and Properties of Actions

<i>(Meta-program)</i> $\mathbb{P}$	$::= (C, I)$	$\llbracket C, a \rrbracket_{\mathcal{M}}^0$	$:= loop$
<i>(Proc)</i> $I$	$::= nil \mid \iota \mid [1] \mid I_1; I_2$ $\mid (b? I_1 + I_2)$	$\llbracket C, nil \rrbracket_{\mathcal{M}}^n$	$:= id$
<i>(Proc Heap)</i> $C$	$::= \{1 \rightsquigarrow I\}^*$	$\llbracket C, I \rrbracket_{\mathcal{M}}^n$	$:= (\mathcal{M}(\iota))$
<i>(Labels)</i> $l$	$::= n$ (nat numbers)	$\llbracket C, [1] \rrbracket_{\mathcal{M}}^n$	$:= \llbracket C, C(1) \rrbracket_{\mathcal{M}}^{n-1}$
<i>(Spec Heap)</i> $\Psi, \mathcal{L}$	$::= \{1 \rightsquigarrow (p, g)\}^*$	$\llbracket C, I; I' \rrbracket_{\mathcal{M}}^n$	$:= \llbracket C, I \rrbracket_{\mathcal{M}}^n \circ \llbracket C, I' \rrbracket_{\mathcal{M}}^n$
		$\llbracket C, (b? I_1 + I_2) \rrbracket_{\mathcal{M}}^n$	$:= \llbracket C, I_1 \rrbracket_{\mathcal{M}}^n \oplus_{\mathcal{M}(b)} \llbracket C, I_2 \rrbracket_{\mathcal{M}}^n$

**Fig. 6.** Syntax and Semantics of the Meta-Language

(actual code is given in Appendix A of TR[19]). The darker green boxes represent computational models with primitives labeled. The diagram shows how each module of BabyVMM will be certified in the model best suited for it. For example, the high-level kernel is certified in the AS model, meaning that it does not see underlying physical memory at all. The implementation of `as_request` and `as_release` are defined over an abstract page map, and thus do not have to know how the hardware deals with page tables, and so on. The plan also indicates which primitives are implemented by which code (lines with circles). When we certify the code, these will be the cross-abstraction links we will have to prove. Lastly, the plan also indicates the stubs in the initialization, which are needed to certify calls from `init` to functions defined over higher abstraction. The PE and PD models are restrictions on HW model, where AT is always on, and always off respectively. ALD is an analogue of ALE, where AT is off.

On boot, the AT is off, and `init` is called. The `init` then calls `mem_init` to initialize the allocation table and `pt_init` to initialize the page tables. Then, `init` uses the HW primitives to enable AT, and jumps into the high-level kernel by calling `kernel_init`.

We will now focus on the technical details to put this plan in action.

$$\begin{array}{c}
\frac{\forall l \in \text{dom}(\mathbb{C}). \mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{C}(l) : \Psi(l)}{\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi} \text{ (CODE)} \qquad \frac{\mathcal{M}, \Psi \vdash \mathbb{I} : (p', g') \quad (p, g) \supseteq (p', g')}{\mathcal{M}, \Psi \vdash \mathbb{I} : (p, g)} \text{ (WEAK)} \\
\frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (p', g') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (p'', g'')}{\mathcal{M}, \Psi \vdash (b? \mathbb{I}' + \mathbb{I}'') : (p', g') \oplus_{\mathcal{M}(b)} (p'', g'')} \text{ (SPLIT)} \quad \frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (p', g') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (p'', g'')}{\mathcal{M}, \Psi \vdash \mathbb{I}'; \mathbb{I}'' : ((p', g') \circ (p'', g''))} \text{ (SEQ)} \\
\frac{}{\mathcal{M}, \Psi \vdash \iota : \mathcal{M}(\iota)} \text{ (PERF)} \qquad \frac{}{\mathcal{M}, \Psi \vdash [1] : \Psi(1)} \text{ (CALL)} \qquad \frac{}{\mathcal{M}, \Psi \vdash \text{nil} : id} \text{ (NIL)}
\end{array}$$

Fig. 7. Static Semantics of the Meta-Language

### 3 Certifying with Refinement

Our framework for multi-machine certification is defined in two parts. First, we create a machine-independent verification framework that will allow us to define quickly and easily as many machines for verification as we need. Second, we will develop our notion of refinements which will allow us to link all the separate machines together.

#### 3.1 A Machine-Independent Certification Framework

Our Hoare-logic based framework is parametric over the definition of operational semantics of the machine, and is sound no matter what machine semantics it is parameterized with. To begin defining such a framework, we first need to understand what exactly is a machine on which we can certify code. The definition that we use is given in Figure 4. Our notion of the machine consists of the following parts:

- State type ( $\Sigma$ ). Define the set of all possible states in a machine.
- Operations ( $\mathcal{A}$ ). This is a set of names of all operations that the machine supports. The set can be infinite, and defined parametrically.
- Conditionals ( $\beta$ ). Defines a type of expressions that are used for branching.
- Conditional Interpreter ( $\gamma$ ). Converts conditionals into state predicates.
- The operational semantics  $\text{OS}$ . This is the main portion of the machine definition. It is a set of actions  $(p, g)$  named by all operations in the machine.

The most important bit of information in the machine are the semantics ( $\text{OS}$ ). The semantics of operations are defined by a precondition ( $p$ ), which shows when the operation is safe to execute, and by a state relation ( $g$ ) that defines the set of possible states that the operation may result in. We will refer to the pair of  $(p, g)$  as an action of the operation. Later we will also use actions to define the specification of programs. Because the type of actions is somewhat complex, we define action combinators in Figure 5, including composition and branching. The same figure also shows the weaker than relation between actions.

Although, at this point we have defined our machines, it does not have any notion of computation. To make use of the machine, we will need to define a concept of programs, as well as what it means for the particular program to execute.

The definition of the program is given in Figure 6. The most important definition in that figure is that of the procedure,  $\mathbb{I}$ . The procedure is a bit of program logic that sequences together calls to the operations of a machine ( $\iota$ ), or to other procedures [1] (loops are implemented as recursive calls). Procedures also include a way to branch on a condition. The procedures can be given a name, and placed in the procedure heap  $\mathbb{C}$ , where they can be referenced from other procedures through the [1] constructor. The procedure heap together with a program rest (the currently executing procedure) makes up the program that can be executed.

The meaning of executing a program is given by the indexed denotational semantics shown on the right side of Figure 6. The meaning of the program is an action that is constructed by sequencing operations. As programs can be infinite, the semantics are indexed by the depth of procedure inclusion.

We use the static semantics (Figure 7) to approximate the action of a procedure. These semantics are similar to the denotational semantics of the meta-language, except that the specifications of called procedure are looked up in the table ( $\Psi$ ). This means that the static semantics works by the programmer approximating the actions of (specifying) the program, and then making sure that the actual action of the program is within the specifications. These well-formed procedures are then grouped into a well-formed module using the `CODE` rule, which forms the concept of a certified module  $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$ , where every procedure in  $\mathbb{C}$  is well-formed under specification in  $\Psi$ . The module also defines a library ( $\mathcal{L}$ ) which is a set of specifications of stubs, i.e. procedures that are used by the module, but are not in the module. These stubs can then be eliminated by providing procedures that match the stubs (see Section 3.2). For a program to be completely certified, all stubs must either be considered valid primitives or eliminated.

For a proof of partial correctness, please see the TR.

### 3.2 Linking

When we certify using modules, it will be very common that the module will require stubs for the procedures of another module. Linking two modules together should replace the stubs in both modules for the actual procedures that are now present in the linked code. The general way to accomplish this is by the following linking lemma:

#### Theorem 1 (Linking)

$$\frac{\mathcal{M}, \mathcal{L}_1 \vdash \mathbb{C}_1 : \Psi_1 \quad \mathcal{M}, \mathcal{L}_2 \vdash \mathbb{C}_2 : \Psi_2 \quad \mathbb{C}_1 \perp \mathbb{C}_2 \quad \mathcal{L}_1 \perp \Psi_2 \quad \mathcal{L}_2 \perp \Psi_1 \quad \mathcal{L}_1 \perp \mathcal{L}_2}{\mathcal{M}, ((\mathcal{L}_1 \cup \mathcal{L}_2) \setminus (\Psi_1 \cup \Psi_2)) \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2} \text{ (LINK)}$$

where  $\Psi_1 \perp \Psi_2 \triangleq \forall l \in \text{dom}(\Psi_1). (l \notin \text{dom}(\Psi_2) \vee \Psi_1(l) = \Psi_2(l))$ .

However, the above rule does not always apply immediately. When the two modules are developed independently, it is possible that the stubs of one module are weaker than the specifications of the procedures that will replace the stubs, which breaks the linking lemma. To fix this, we strengthen the library.

#### Theorem 2 (Stub Strengthening)

If  $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$ , then for any  $\mathcal{L}'$  s.t.  $\forall l \in \text{dom}(\mathcal{L}). \mathcal{L}(l) \supseteq \mathcal{L}'(l)$  and  $\text{dom}(\mathcal{L}') \cap \text{dom}(\Psi) = \emptyset$ , the following holds:  $\mathcal{M}, \mathcal{L}' \vdash \mathbb{C} : \Psi$ .

This theorem allows us to strengthen the stubs to match the specs of procedures, enabling the linking lemma. Of course, if the specs of the real procedures are not stronger than the specs of the stubs, then the procedures do not correctly implement what the module expects, and linking is not possible.

### 3.3 The Refinement Framework

Up to this point, we have only considered what happens to the code that is certified over a single machine. However, the purpose of our framework is to facilitate multi-machine verification. For this purpose, we construct the refinement framework that will allow us to refine certified modules in one machine to certified modules in another. The most general notion of refinement in our framework can be defined by the following:

#### Definition 1 (Certified Refinement)

A certified refinement from machine  $\mathcal{M}_A$  to machine  $\mathcal{M}_C$  is a pair of relations  $(T_C, T_\psi)$  and a predicate over the abstract certified module  $Acc$ , such that for all  $\mathbb{C}_A, \Psi'_A, \Psi_A$ , the following holds

$$\frac{\mathcal{M}_A, \Psi'_A \vdash \mathbb{C}_A : \Psi_A \quad Acc(\mathcal{M}_A, \Psi'_A \vdash \mathbb{C}_A : \Psi_A)}{\mathcal{M}_C, T_\psi(\Psi'_A) \vdash T_C(\mathbb{C}_A) : T_\psi(\Psi_A)} \text{REFINE}$$

This definition is not a rule, but a template for other definitions. To define a refinement, one has to provide the particular  $T_C, T_\psi, Acc$  together with the proof that the rule holds. However, instead of trying to define these translations directly, we will automatically generate them from the relations between the particular pairs of machines.

**Representation Refinement.** The only automatic refinement we will discuss in this paper is the representation refinement. The representation refinement can be generated for an abstract ( $\mathcal{M}_A$ ) and a concrete machine ( $\mathcal{M}_C$ ), where both use the same operations and conditionals (e.g.  $\mathcal{M}_A.\Delta = \mathcal{M}_C.\Delta$  and  $\mathcal{M}_A.\beta = \mathcal{M}_C.\beta$ ) by defining a relation ( $\text{repr} : \mathcal{M}_A.\Sigma \rightarrow \mathcal{M}_C.\Sigma \rightarrow Prop$ ) between the states of the two machines. Using  $\text{repr}$ , we can define our specification translation function:

$$T_{A-C}(p, g) \triangleq (\lambda \mathbb{S}_C. \exists \mathbb{S}_A. \text{repr } \mathbb{S}_A \mathbb{S}_C \wedge p \mathbb{S}_A, \lambda \mathbb{S}_C. \lambda \mathbb{S}'_C. \forall \mathbb{S}_A. \text{repr } \mathbb{S}_A \mathbb{S}_C \rightarrow \forall \mathbb{S}'_A. g \mathbb{S}_A \mathbb{S}'_A \rightarrow \text{repr } \mathbb{S}'_A \mathbb{S}'_C)$$

This operation creates an concrete action from an abstract action. Informally it works as follows. There must be at least one abstract state related to the starting concrete state for which the abstract action applies. The action starting from state  $\mathbb{S}_C$  results in set containing  $\mathbb{S}'_C$ , only if for all related abstract states for which the abstract action is valid result in sets of abstract states that contain a state related to  $\mathbb{S}'_C$ . Essentially, the resulting concrete action is an intersection of all abstract actions that do not fail.

To make this approach work, we require several properties over the machines and the  $\text{repr}$ . First, the refined semantics of abstraction operations have to be weaker than the semantics of their concrete counterparts, e.g.  $\forall \iota_A \in \mathcal{M}_A. T_{A-C}(\mathcal{M}_A(\iota_A)) \supseteq \mathcal{M}_C(\iota_A)$ .

Second, the refinement must preserve the branch choice, e.g. if the refined program chooses left branch, then abstract program had to choose the left branch in all states related by  $\text{repr}$  as well. This property is ensured by requiring the following:

$$\forall b. \forall \mathbb{S}, \mathbb{S}'. (\exists \mathbb{S}_C. \text{repr}(\mathbb{S}, \mathbb{S}_C) \wedge \text{repr}(\mathbb{S}', \mathbb{S}_C)) \rightarrow (\mathcal{M}(b) \mathbb{S} \leftrightarrow \mathcal{M}(b) \mathbb{S}')$$

With these properties, we can define a valid refinement by the following lemma:

**Lemma 1 (repr-refinement valid)**

Given `repr` with proofs of the two properties above, the following is valid:

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi_A}{\mathcal{M}_C, T_\Psi(\mathcal{L}_A) \vdash \mathbb{C} : T_\Psi(\Psi_A)}$$

where  $T_\Psi(\Psi) := \{T_{A-C}(\Psi(1)) \mid 1 \in \text{dom}(\Psi)\}$

This refinement is interesting in that it preserves the code of the program, and performing point-wise refinement on specifications. Our actual work defines several other refinement generators. One of these, code-preserving refinement, is included in the TR, and is used as a stepping stone for proof of Lemma 1. Coq implementation features more general versions of refinements presented, as well as several others.

## 4 Certifying C Code

Since BabyVMM is written in C, we define a formal specification of a tiny subset of the C language using our framework. This C machine will be parameterized by the specific semantics of the memory model, as our plan required. We will also utilize the C machine to further speed up the creation of refinements.

### 4.1 The Semantics of C

To define our C machine in terms of our verification framework, we need to give it a state type, a list of operations, and the semantics of those operations expressed as actions. All of these are given in Figure 8.

The state of the C machine includes two components, the stack and the memory. The stack is an abstract C stack that consists of a list of frames, which include call, data, and return frames. In the current version, the stack is independent from memory (one can think of it existing within a statically defined part of the loaded kernel). The memory model is a parameter in the C machine, meaning that it can make use of any memory model as long as it defines load and store operations. The syntax of the C machine is different from the usual definition, in that it relies on the meta-machine for its control flow by using the meta-machines `call` and `branch`. Our definition of C adds atomic operations that perform state updates. Thus the operations include two types assignments - one to stack and one to memory, and 4 operations to manipulate stack for call and return, which push and pop the frames.

Because control flow is provided by a standard machine, the code has to be modified slightly. For example, a function call of the form  $r = f(x)$  will split into a sequence of three operations: `fcall([x]); [f]; readret([r])`, the first setting up a call frame, the second making the call, and the third doing the cleanup. Similarly, the body of the function  $f(x)\{body; return(0); \}$  will become `args([x]); body; ret(0)`, as the function must first



move the arguments from the call frame into a data frame. Loops have to be desugared into recursive procedures with branches. These modifications are entirely mechanical, and hence we can claim that our machine supports desugared linearized C code.

## 4.2 Refinement in C Machines

C machines at different abstraction layers differ only in their memory models, with the stack being the same. We can use this fact to generate refinements between the C machines using only the representation relation between memory models. This relation ( $M_1 \leq M_2$ ) can be completely arbitrary as long as these conditions hold:

- (State)  $\mathbb{S} ::= (M, S)$   
(Memory)  $M ::= (\text{any type over which } load(M, l) \text{ and } store(M, l, w) \text{ are defined})$   
(Stack)  $S ::= nil \mid Call(list\ w) :: S \mid Data(\{v \rightsquigarrow w\} :: S) \mid Ret(w) :: S$   
(Expressions)  $e ::= se \mid *(e)$   
(StackExpr, Cond)  $se, b ::= w \mid v \mid binop(bop, e_1, e_2)$   
(Binary Operators)  $bop ::= + \mid - \mid * \mid / \mid \% \mid == \mid < \mid <= \mid >= \mid > \mid != \mid \&\& \mid ||$   
(Variables)  $v ::= (\text{a decidable set of names})$   
(Words)  $w ::= n$  (integers)  
(Operation)  $\iota ::= v := e \mid *(e_{loc}) := e \mid fcall(list\ e) \mid ret(e) \mid args(list\ v) \mid readret(v)$

Operation ( $\iota$ ) =	Action ( $M(\iota)$ ) =
$v := e$	$(\lambda \mathbb{S}. \exists S', F, w. \mathbb{S}. S = Data(F) :: S' \wedge eval(e, \mathbb{S}) = w,$ $\lambda \mathbb{S}, S'. \exists S', F, w. \mathbb{S}. S = Data(F) :: S' \wedge eval(e, \mathbb{S}) = w \wedge$ $\mathbb{S}'. M = \mathbb{S}. M \wedge S'. S = Data(F\{v \rightsquigarrow w\}) :: S')$
$*(e_{loc}) := e$	$(\lambda \mathbb{S}. \exists l, w. eval(e, \mathbb{S}) = w \wedge eval(e_{loc}, \mathbb{S}) = l \wedge \exists M'. M' = store(M, l, w),$ $\lambda \mathbb{S}, S'. \exists l, w. eval(e, \mathbb{S}) = w \wedge eval(e_{loc}, \mathbb{S}) = l \wedge$ $\mathbb{S}'. M = store(\mathbb{S}. M, l, w) \wedge S'. S = \mathbb{S}. S)$
$fcall([e_1, \dots, e_n])$	$(\lambda \mathbb{S}. \exists v_1, \dots, v_n. eval(e_1, \mathbb{S}) = v_1 \wedge \dots \wedge eval(e_n, \mathbb{S}) = v_n,$ $\lambda \mathbb{S}, S'. \exists v_1, \dots, v_n. eval(e_1, \mathbb{S}) = v_1 \wedge \dots \wedge eval(e_n, \mathbb{S}) = v_n \wedge$ $\mathbb{S}'. M = \mathbb{S}. M \wedge S'. S = Call([v_1, \dots, v_n]) :: \mathbb{S}. S)$
$args([v_1, \dots, v_n])$	$(\lambda \mathbb{S}. \exists w_1, \dots, w_n, S'. \mathbb{S}. S = Call([w_1, \dots, w_n]) :: S',$ $\lambda \mathbb{S}, S'. \exists w_1, \dots, w_n, S'. \mathbb{S}. S = Call([w_1, \dots, w_n]) :: S' \wedge$ $\mathbb{S}'. M = \mathbb{S}. M \wedge S'. S = Data(\{v_1 \rightsquigarrow w_1, \dots, v_n \rightsquigarrow w_n\}) :: S')$
$readret(v)$	$(\lambda \mathbb{S}. \exists S', w. \mathbb{S}. S = Ret(w) :: Data(D) :: S',$ $\lambda \mathbb{S}, S'. \exists S', w. \mathbb{S}. S = Ret(w) :: Data(D) :: S' \wedge$ $\mathbb{S}'. M = \mathbb{S}. M \wedge S'. S = Data(D\{v \rightsquigarrow w\}) :: S')$
$ret(e)$	$(\lambda \mathbb{S}. \exists w. eval(e, \mathbb{S}) = w, \lambda \mathbb{S}, S'. \mathbb{S}'. M = \mathbb{S}. M \wedge S'. S = Ret(eval(e, \mathbb{S})) :: \mathbb{S}. S)$

$$eval(e, \mathbb{S}) ::= \begin{cases} w & \text{if } e = w \\ \mathbb{S}. S(v) & \text{if } e = v \\ load(\mathbb{S}. M, eval(e_1, \mathbb{S})) & \text{if } e = *(e_1) \\ b(eval(e_1, \mathbb{S}), eval(e_2, \mathbb{S})) & \text{if } e = binop(b, e_1, e_2) \end{cases}$$

$$\mathcal{T}(b) ::= \lambda \mathbb{S}. eval(b, \mathbb{S}) \neq 0$$

Fig. 8. Primitive C-like machine

Definition	Value	Description
PGSIZE	4096	Number of bytes per page
NPAGES	unspecified	Number of phys. pages in memory
VPAGES	unspecified	Maximum page number of a virtual address
$\text{Pg}(addr)$	$addr/\text{PGSIZE}$	gets page of address
$\text{Off}(addr)$	$addr\%\text{PGSIZE}$	offset into page of address
$\text{LowPg}(pg)$	$0 \leq pg < \text{NPAGES}$	valid page in low memory area
$\text{HighPg}(pg)$	$\text{NPAGES} \leq pg < \text{VPAGES}$	valid page in high memory area

**Fig. 9.** Page Definitions

$$\begin{aligned} \forall l, v. \text{load}(M_1, l) = v \rightarrow \text{load}(M_2, l) = v \\ \forall l, v, M'_1. (M'_1 = (\text{store}(M_1, l, v))) \rightarrow (M'_1 \leq (\text{store}(M_2, l, v))) \end{aligned}$$

The above properties make sure that the load and store operations of memory behave in a similar way. We construct the  $\text{repr}$  between C machine as follows:

$$\text{repr} := \lambda \mathbb{S}_A, \mathbb{S}_C. (\mathbb{S}_A.S = \mathbb{S}_C.S) \wedge (\mathbb{S}_A.M \leq \mathbb{S}_C.M)$$

Using the properties of load and store, we show properties needed for  $\text{repr}$ -refinement to work: that for every operation  $\iota$  in the C machine  $T_{M_1-M_2}(\mathcal{M}_{M_1}(\iota)) \supseteq \mathcal{M}_{M_2}(\iota)$ , and that  $\text{repr}$  preserves branching. For details, please see the TR. Now we can define the actual refinement rule for C machines:

### Corollary 1 (C Refinement)

For any two memory models  $M_1$  and  $M_2$ , s.t.  $M_1 \leq M_2$ , the following refinement works for C machines instantiated with  $M_1$  and  $M_2$ .

$$\frac{\mathcal{M}_{M_1}, \mathcal{L} \vdash \mathbb{C} : \Psi}{\mathcal{M}_{M_2}, T_{M_1-M_2}(\mathcal{L}) \vdash \mathbb{C} : T_{M_1-M_2}(\Psi)} M_1 - M_2$$

Thus we know that if we have two C-machines that have related memory models, then we have a working refinement between the two machines. Our next step is the to show the relations between all the memory models shown in our plan (in Figure 3).

## 5 Virtual Memory Manager

At this point, we have all the machinery necessary to start building our certified memory manager according to the plan. The first step is to formally define and give relations between the memory models that we will use in our certification. Then we will certify the code of the modules that make up the VMM. These modules will then be refined and linked together, resulting in the conclusion that the entire BabyVMM is certified.

### 5.1 The Memory Models

Because of the space limit, we will only formally present the PMAP memory model (Figures 9 and 10). For the definitions of others, please see the TR.

(Global Storage System)  $M ::= (D, A, PM)$   
 (Allocatable Memory)  $D ::= \{addr \rightsquigarrow w \mid \text{LowPg}(\text{Pg}(addr)) \wedge addr \% 8 = 0\}^*$   
 (Page Allocation Table)  $A ::= \{pg \rightsquigarrow \text{bool} \mid \text{LowPg}(pg)\}^*$   
 (Page Map)  $PM ::= \{pg \rightsquigarrow pg' \mid \text{HighPg}(pg)\}^*$

Notation	Definition
$load(M, va)$	$M.D(\text{trans}(M, va))$ if $M.A(\text{Pg}(\text{trans}(M, va))) = true$
$store(M, va, w)$	$(M.D\{\text{trans}(M, va) \rightsquigarrow w\}, M.A, M.PM)$ if $M.A(\text{Pg}(\text{trans}(M, va))) = true$

$$\text{trans}(M, va) := \begin{cases} M.PM(\text{Pg}(va)) * \text{PGSIZE} + \text{Off}(va) & \text{if HighPg}(\text{Pg}(va)) \\ va & \text{otherwise} \end{cases}$$

Label	Specification
mem_alloc	$(\lambda \mathbb{S}. \exists S'. \mathbb{S}.S = \text{Call}([\ ] :: S',$ $\lambda \mathbb{S}, S'. \exists S'. (\mathbb{S}.S = \text{Call}([\ ] :: S') \wedge ((S'.S = \text{Ret}(0) :: S' \wedge S'.M = \mathbb{S}.M) \vee$ $(\exists pg. S'.S = \text{Ret}(pg) :: S' \wedge S'.M.A = \mathbb{S}.M.A[pg \rightsquigarrow true] \wedge S'.M.PM = \mathbb{S}.M.PM \wedge$ $\wedge \mathbb{S}.M.A(pg) = false \wedge \forall l. \mathbb{S}.M.A(\text{Pg}(l)) = true \rightarrow (S'.M.D(l) = \mathbb{S}.M.D(l))))$
mem_free	$(\lambda \mathbb{S}. \exists S', pg. \mathbb{S}.S = \text{Call}([pg]) :: S' \wedge \mathbb{S}.M.A(pg) = true,$ $\lambda \mathbb{S}, S'. \exists S', pg. \mathbb{S}.S = \text{Call}([pg]) :: S' \wedge S'.S = \text{Ret}(0) :: S' \wedge S'.M.PM = \mathbb{S}.M.PM \wedge$ $S'.M.A = \mathbb{S}.M.A[pg \rightsquigarrow false] \wedge \forall l. S'.M.A(\text{Pg}(l)) = true \rightarrow S'.M.D(l) = \mathbb{S}.M.D(l))$
pt_set	$(\lambda \mathbb{S}. \exists S', vp, pp. \mathbb{S}.S = \text{Call}([vp, pp]) :: S' \wedge \text{HighPg}(vp) \wedge \text{LowPg}(pp)$ $\lambda \mathbb{S}, S'. \exists S', vp, pp. \mathbb{S}.S = \text{Call}([vp, pp]) :: S' \wedge S'.S = \text{Ret}(0) :: S' \wedge S'.M.A = \mathbb{S}.M.A \wedge$ $S'.M.PM = \mathbb{S}.M.PM[vp \rightsquigarrow pp] \wedge \forall l. S'.M.A(\text{Pg}(l)) = true \rightarrow S'.M.D(l) = \mathbb{S}.M.D(l))$
pt_lookup	$(\lambda \mathbb{S}. \exists S', vp. \mathbb{S}.S = \text{Call}([vp]) :: S' \wedge \text{HighPg}(vp),$ $\lambda \mathbb{S}, S'. \exists S', vp. \mathbb{S}.S = \text{Call}([vp]) :: S' \wedge S'.S = \text{Ret}(\mathbb{S}.M.PM(vp)) :: S' \wedge S'.M = \mathbb{S}.M)$

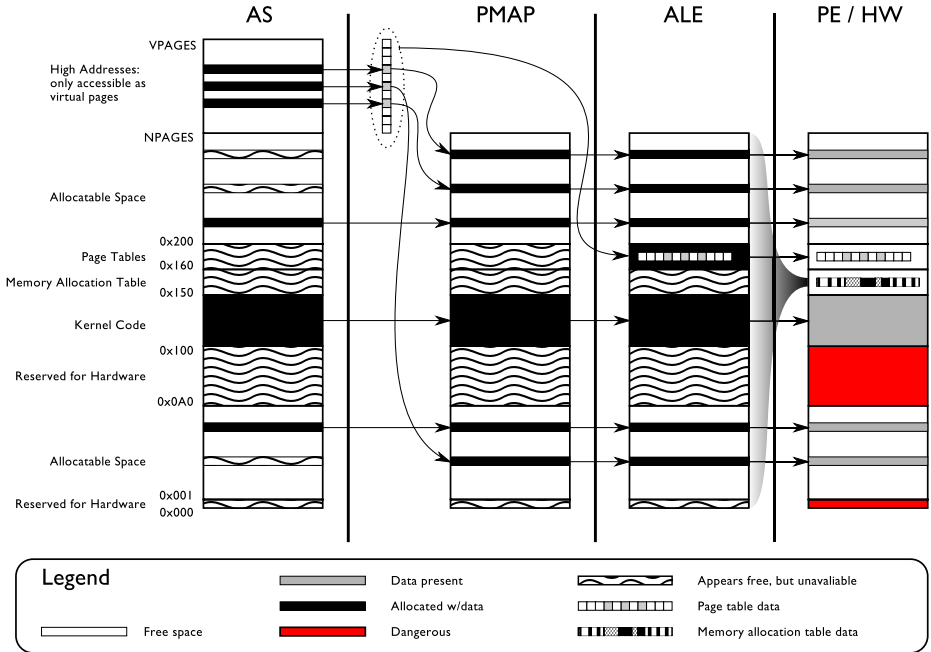
**Fig. 10.** PMAP Memory Model ( $M_{PMAP}$ ) and Library ( $\mathcal{L}_{PMAP}$ )

The state of the PMAP memory has three components, the actual memory store  $D$ , the allocation table  $A$ , and the first-class pagemap  $PM$ . The memory store contains the actual data in memory, indexed by physical addresses. The allocation table  $A$ , keeps track of which pages are allocated and which are not. This allocation information is abstract - it does not have to correspond to the actual allocation table used within the VMM. For example, the hardware page tables, which this model abstracts, are still in memory, but are hidden by the allocation table. The page map is the abstract mapping of virtual pages to physical pages, which purposefully skips all addresses mappable to physical memory. This mapping is used in loads and stores of the memory model, which use the  $\text{trans}$  predicate to translate addresses by looking up mappings in the  $PM$ .

The PMAP model relies on the stub library ( $\mathcal{L}_{PMAP}$ ) for updating auxiliary data structures. There are two stubs for memory allocation, `mem_alloc` and `mem_free`. Their specs show how they modify the allocation table, and how allocating a page is non-deterministic and may potentially return any free page. The other two stubs, `pt_set` and `pt_lookup` update and look up page map entries; their specs are straightforward.

## 5.2 Relation between Memory Models

Our plan calls for creation of the refinements between the memory models. In Section 4.2, we have shown that we can generate a valid refinement by creating a relation



**Fig. 11.** Relation between Memory Models

between the memory states, and then showing that abstract loads and stores are preserved by this relation. These relations and proofs of preserving the memory operations are fairly lengthy and quite technical, and thus we leave the mathematical detail to our Coq implementation, opting for a visual description shown in Figure 11.

On the right is a state of the hardware memory, whose operational semantics gives little protection from accessing data. Some areas of memory are dangerous, some are empty, others contain data, including the allocation tables and page tables. This memory relates to the ALE memory model by abstracting out the memory allocation table. This allocation table now offers protection for accessing both the unallocated space, and the space that seems unallocated, but dangerous to use (marked by wavy lines). An example of such area is the allocation table itself - the ALE model hides the table, making it appear to be unusable. The ALE `mem_alloc` primitive will never allocate pages from these wavy areas, protecting them without complicating the memory model.

The relation between the PMAP and ALE models shows that the abstract pagemap of PMAP model is actually contained within the specific area of the ALE model. The relation makes sure that the mappings contained in the PMAP's pagemap are the same as the translation results of the ALE's page table structures. To protect the in memory page tables, the relation hides the page table memory area from the PMAP model, using the same trick as the one used to protect the allocation tables in the ALE model.

The relation between the AS and PMAP models collapses PMAP's memory and the page maps into a single memory like structure in the AS model. This is mostly

accomplished by chaining the translation mechanism with the storage mechanism. However, to make this work, it is imperative that the relation ensures that no two pages of the AS model ever map to the same physical page in the PMAP model. This means that all physical pages that are mapped from the high-addresses become hidden in the AS model. We will not go into detail about the preservation of load and stores, as these proofs are mostly straightforward, given the relations.

### 5.3 Certification and Linking of BabyVMM

We have verified all the functions of the virtual memory on the appropriate memory models. This means that we have defined appropriate specifications for our functions, and certified our code. We also make an assumption that a kernel is certified in the AS model. The result is the following certified modules:

$$\begin{array}{lll} \mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem} : \Psi_{PE}^{mem} & \mathcal{M}_{ALE}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi_{PMAP}^{as} & \mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{meminit} : \Psi_{PD}^{meminit} \\ \mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi_{ALE}^{pt} & \mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi_{AS}^{kernel} & \mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{ptinit} : \Psi_{ALD}^{ptinit} \end{array}$$

However, the `init` function makes calls to other procedures that are certified in more abstract machines. Thus to certify `init` over the  $\mathcal{M}_{HW}$  machine, we will need to create stubs for these procedures, which have to be carefully crafted to be valid for the refined specifications of the actual procedures. Thus, the specification of `init` results in the following:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \left\{ \text{kernel\_init} \rightsquigarrow \mathfrak{a}_{HW}^{kernel-init}, \text{mem\_init} \rightsquigarrow \mathfrak{a}_{HW}^{meminit}, \text{pt\_init} \rightsquigarrow \mathfrak{a}_{HW}^{ptinit} \right\} \vdash \mathbb{C}^{init} : \Psi_{HW}^{init}$$

With all the modules verified, we proceed to link them together. The first step is to refine the kernel. We use our AS-PMAP refinement rule to get the refined module:

$$\mathcal{M}_{PMAP}, T_{AS-PMAP}(\mathcal{L}_{AS}) \vdash \mathbb{C}^{kernel} : T_{AS-PMAP}(\Psi_{AS}^{kernel})$$

Then we show that the specs of functions and the primitives of the PMAP machine are proper implementation of the refined specs of  $\mathcal{L}_{AS}$ , more formally,  $T_{AS-PMAP}(\mathcal{L}_{AS}) \supseteq \mathcal{L}_{PMAP} \cup \Psi_{PMAP}^{as}$ . Using library strengthening and the linking lemma, we produce a certified module that is the union of the refined kernel and address space library:

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} : T_{AS-PMAP}(\Psi_{AS}^{kernel}) \cup \Psi_{PMAP}^{as}$$

Applying this process to all the modules over all refinements, we link all parts of the code, except `init` certified over  $\mathcal{M}_{HW}$ . For readability, we hide chains of refinements. For example,  $T_{AS-HW}$  is actually  $T_{AS-PMAP} \circ T_{PMAP-ALE} \circ T_{ALE-PE} \circ T_{PE-HW}$ .

$$\begin{aligned} \mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash & \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} : \\ & T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PMAP-HW}(\Psi_{PMAP}^{as}) \cup T_{ALE-HW}(\Psi_{ALE}^{pt}) \cup \\ & T_{PE-HW}(\Psi_{PE}^{mem}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \end{aligned}$$

To get the initialization to link with the refined module, we must make sure that the stubs that we have developed for `init` are compatible with the refined specifications of the actual functions. This means that we prove the following:

$$\begin{aligned} a_{HW}^{kernel-init} &\supseteq T_{AS-HW}(\Psi_{AS}^{kernel})(kernel-init) \\ a_{HW}^{meminit} &\supseteq T_{PD-HW}(\Psi_{PD}^{meminit})(mem-init) & a_{HW}^{ptinit} &\supseteq T_{ALD-HW}(\Psi_{ALD}^{ptinit})(pt-init) \end{aligned}$$

Using these properties, we apply stub strengthening to the init module:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \vdash \mathbb{C}^{init} : \Psi_{HW}^{init}$$

This certification is now linkable to the rest of the VMM and kernel, to produce the final result that we need:

$$\begin{aligned} \mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash &\mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} \cup \mathbb{C}^{init} : \\ &T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PMAP-HW}(\Psi_{PMAP}^{as}) \cup T_{ALE-HW}(\Psi_{ALE}^{pt}) \cup \\ &T_{PE-HW}(\Psi_{PE}^{mem}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \cup \Psi_{HW}^{init} \end{aligned}$$

This result means that given a certified kernel in the AS model, we can refine it to the HW model of memory by linking it with VMM implementation. Furthermore, it is safe to start this kernel by calling the `init` function, which will perform the setup, and then call the `kernel-init` function, the entry point of the high-level kernel.

## 6 Coq Implementation

All portions of this system have been implemented in the Coq Proof Assistant[5]. The portions of the implementation directly related to the BabyVMM verification, including C machines, refinements, specs, and related proofs (excluding frameworks) took about 3 person-months to verify. The approximate line counts for unoptimized proof are:

- Verification and refinement framework - 3000 lines
- Memory models - 200-400 lines each
- `repr` and compatibility between models - 200-400 lines each
- Compatibility of stubs and implementation - 200-400 lines per procedure
- Code verification - less than 200 lines per procedure (half of it boilerplate).

## 7 Related Work and Conclusion

The work presented here is a continuation of the work on Hoare-logic frameworks for verification of system software. The verification framework evolved from SCAP[8] and GCAP[3]. Although our framework does not mention separation logic[17], information hiding[16], and local action[4] explicitly, these methods had great influence on the design of the meta-language and the refinements. The definition of `repr` generalizes the work on certified garbage collector[15] to fit our concept of refinement. The project's motivation is the modular and reusable certification of the CertiKOS kernel[10].

The well-known work in OS verification is L4.verified[12,6], which has shown a complete verification of an OS kernel. Their methodology is different, but they have

considered verification of virtual memory [13,14]. However, their current kernel verification does not abstract virtual memory, maintaining only the invariant that allows the kernel to function, and leaving the details to the user level.

The Verisoft project [9,2,11,11,18] is the work that is closest to ours. We both aim for pervasive verification of OS by doing foundational verification of all components. Both works utilize multiple machines, and require linking. As both projects aim for certification of a kernel, both have to handle virtual memory. Although Verisoft uses multiple machine models, they use them sparingly. For example, the entire microkernel, excluding assembly code, is specified in a single layer, with correctness shown as a single simulation theorem between the concurrent user thread model (CVM) and the instruction set. The authors mention that the proof of correctness is a more complex part of Verisoft. Such monolithic approach is susceptible to local modifications, where a small change in one part of microkernel may require changes to the entire proof.

Our method for verification defines many more layers, with smaller refinement proofs between them, and composes them to produce larger abstractions, ensuring that the verification is more reusable and modular. Our new framework enables us to create abstraction layers with less overhead, reducing the biggest obstacle to our approach. We have demonstrated the practicality of our approach by certifying BabyVMM, a small virtual memory manager running on simplified hardware, using a new layer for every non-trivial abstraction we could find.

**Acknowledgements.** We thank anonymous referees for suggestions and comments on an earlier version of this paper. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0910670 and 1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

## References

1. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load: Leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: OS Verification* 42, 389–454 (2009)
2. Alkassar, E., Schirmer, N.W., Starostin, A.: Formal Pervasive Verification of a Paging Mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)
3. Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: *Proc. PLDI 2007*, pp. 66–77. ACM, New York (2007)
4. Calcagno, C., O’Hearn, P., Yang, H.: Local action and abstract separation logic. In: *Proc. LICS 2007*, pp. 366–378 (July 2007)
5. Coq Development Team. *The Coq proof assistant reference manual*. The Coq release v8.0 (October 2005)
6. Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., Heiser, G.: Towards a practical, verified kernel. In: *Proc. HoTOS 2007*, San Diego, CA, USA (May 2007)
7. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: *Proc. PLDI 2008*, pp. 170–182. ACM (2008)

8. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. In: PLDI 2006, pp. 401–414 (June 2006)
9. Gargano, M., Hillebrand, M.A., Leinenbach, D., Paul, W.J.: On the Correctness of Operating System Kernels. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 1–16. Springer, Heidelberg (2005)
10. Gu, L., Vaynberg, A., Ford, B., Shao, Z., Costanzo, D.: Certikos: A certified kernel for secure cloud computing. In: Proc. APSys 2011. ACM (2011)
11. In der Rieden, T.: Verified Linking for Modular Kernel Verification. PhD thesis, Saarland University, Computer Science Department (November 2009)
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proc. SOSP 2009, pp. 207–220 (2009)
13. Klein, G., Tuch, H.: Towards verified virtual memory in I4. In: TPHOLs Emerging Trends 2004, Park City, Utah, USA (September 2004)
14. Kolanski, R., Klein, G.: Mapped Separation Logic. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 15–29. Springer, Heidelberg (2008)
15. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Proc. PLDI 2007, pp. 468–479 (2007)
16. O’Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL 2004, pp. 268–280 (January 2004)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. LICS 2002, pp. 55–74 (July 2002)
18. Starostin, A.: Formal Verification of Demand Paging. PhD thesis, Saarland University, Computer Science Department (March 2010)
19. Vaynberg, A., Shao, Z.: Compositional verification of BabyVMM (extended version and Coq proof). Technical Report YALEU/DCS/TR-1463, Yale University (October 2012), <http://flint.cs.yale.edu/publications/babyvmm.html>



# Shall We Juggle, Coinductively?

Keisuke Nakano

The University of Electro-Communications, Japan  
ksk@cs.uec.ac.jp

**Abstract.** Buhler et al. presented a mathematical theory of toss juggling by regarding a toss pattern as an arithmetic function, where the function must satisfy a condition for the pattern to be valid. In this paper, the theory is formalized in terms of *coinduction*, reflecting the fact that the validity of toss juggling is related to a property of infinite phenomena. A tactic is implemented for proving the validity of toss patterns in Coq. Additionally, the completeness and soundness of a well-known algorithm for checking the validity is demonstrated. The result exposes a practical aspect of coinductive proofs.

## 1 Introduction

Toss juggling, the most popular form of juggling, involves the tossing of multiple objects (e.g., balls, clubs, rings, knives, torches, and pieces of fruit) into the air and catching them by hand in succession. It has been not only artistically impressive to people in the audience but also theoretically attractive to many mathematicians like Claude E. Shannon and Ronald Graham because of the scientific features of its patterns.

A pattern can be represented by a finite sequence of non-negative integers, called *siteswap* [BEGW94]. Each integer in a siteswap corresponds to a beat and encodes the height of the toss, i.e., the number of beats for which the tossed object is in the air. Consider a very simple siteswap  $\langle 3.0.0 \rangle$ , which represents trivial 1-object juggling. The 3 means tossing an object so that it stays in the air for 3 beats. The two 0's mean no object is caught and tossed. Since a finitely represented siteswap implicitly represents an infinite toss pattern due to repetition,  $\langle 3.0.0 \rangle$  actually denotes toss pattern  $\langle 3.0.0.3.0.0.3.0.0.3.0.0 \dots \rangle$ . We assume that an object (possibly nothing) is tossed by an alternate hand on every beat. Suppose that we start by tossing an object with the right hand (R), as shown in Fig. 1. Since it stays in the air for 3 beats, it falls into the left hand (L) and

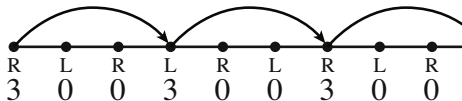


Fig. 1. Siteswap  $\langle 3.0.0 \rangle$

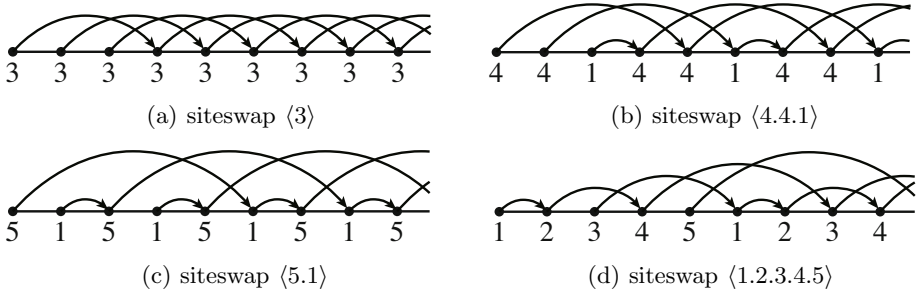


Fig. 2. Examples of valid siteswap patterns

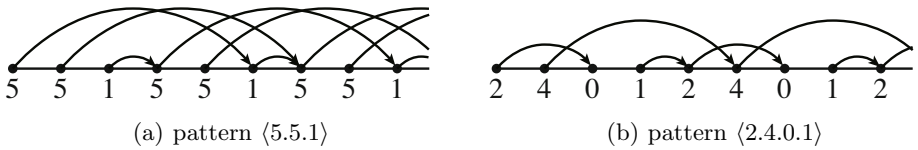


Fig. 3. Examples of invalid patterns

is thrown up again 3 beats later, to be caught in the right hand, and so on. A 0 between two 3's denotes a rest beat for each hand during which they do not catch, toss, or hold an object.

Figure 2 shows four diagrams generated using siteswaps in the labels. Siteswap  $\langle 3 \rangle$  denotes infinite pattern  $\langle 3.3.3 \dots \rangle$ , which represents a standard pattern of 3-ball juggling, called a cascade. Siteswap  $\langle 4.4.1 \rangle$  denotes infinite pattern  $\langle 4.4.1.4.4.1 \dots \rangle$ , in which no two objects will be caught simultaneously by the same hand as shown in the figure. We can also check such validity of  $\langle 5.1 \rangle$  and  $\langle 1.2.3.4.5 \rangle$  in a similar way. The number of orbits in the figure corresponds to the number of juggled objects.

Note that not all finite sequences of non-negative integers are valid siteswaps. For instance,  $\langle 5.5.1 \rangle$  and  $\langle 2.4.0.1 \rangle$  are not valid, as easily proved by drawing their diagrams. As shown in Fig. 3, the two objects tossed on the second and sixth beats in the  $\langle 5.5.1 \rangle$  pattern are caught simultaneously by the same hand<sup>1</sup>. In the  $\langle 2.4.0.1 \rangle$  pattern, the object tossed on the first beat is caught two beats later by a hand that should be at rest for that beat. Since an orbit corresponds to an object's movement, this toss pattern represents extinguishing the object. In short, these patterns are not valid siteswaps.

On the other hand, how can we prove the validity of the  $\langle 4.4.1 \rangle$  siteswap? No matter how far we extend the diagram in Fig. 2(b), we cannot prove that

<sup>1</sup> There is a siteswap notation for multiplex toss juggling that allows for a hand to manipulate two objects at the same time. In this paper, we deal with the standard (vanilla) siteswap notation in which each hand manipulates at most one object at a time.

two objects never collide with each other. We need an effective technique for completing a proof in finite steps.

Our approach to meeting this need is based on the use of *coinduction* to formalize the ‘never collide’ property. While most applications of coinduction in computer science are related to bisimulation [San09], reasoning about infinite behaviors is also a natural application of coinduction [LG09, NU09]. In Section 2, we start by formalizing a theory for validating toss patterns in terms of coinduction and validate specific toss patterns by coinduction. We also present a general tactic for validating arbitrary patterns. In Section 3, we describe the implementation of a well-known algorithm [BEGW94] for validating siteswap patterns.

All theorems in this paper are certified in Coq and the source code is available at <http://millsmess.cs.uec.ac.jp/~ksk/siteswap/>. Coq-style syntax is used for formal definitions in the rest of this paper.

## 2 Toss Streams and Their Validity

We formalize a theory for validating juggling toss patterns by using a *stream* which is a typical coinductive structure, because a toss pattern has an infinite length. The definition of pattern validity is also given by using coinduction. We will show several examples of proofs for validating patterns and present a general tactic for validation.

### 2.1 Toss Streams

A stream of succeeding tosses, a *toss stream*, is defined by

```
CoInductive toss : Set := Toss : nat -> toss -> toss.
Infix "::~" := Toss (right associativity, at level 60).
```

in which each number (non-negative integer) encodes the height of the toss. The second line introduces a right associative infix symbol `::~` as an alias of the `Toss` constructor. For example, we can define  $\langle 3.0.0 \rangle$ ,  $\langle 4.4.1 \rangle$ , and  $\langle 1.2.3.4.5 \rangle$  by

```
CoFixpoint toss_300 : toss := 3 ::~ 0 ::~ 0 ::~ toss_300.
CoFixpoint toss_441 : toss := 4 ::~ 4 ::~ 1 ::~ toss_441.
CoFixpoint toss_12345 : toss :=
  1 ::~ 2 ::~ 3 ::~ 4 ::~ 5 ::~ toss_12345.
```

respectively, by using a co-recursive definition. Hereafter,  $\langle 4.4.1 \rangle$  represents an infinite repetition of 4, 4, and 1. We will implicitly use equivalence between  $\langle 4.4.1 \rangle$  and  $4 ::~ \langle 4.1.4 \rangle$ , which can be proved by coinduction.

We use  $t_n$  to represent the height of the  $n$ -th toss of a toss stream  $t$ , where the stream starts with the 0-th toss; i.e., toss stream  $t$  is equivalent to  $t_0 ::~ t_1 ::~ t_2 ::~ \dots$ . An object tossed on the  $n$ -th beat is caught on the  $(t_n + n)$ -th beat.

$$\begin{array}{c}
 \frac{\vdash t}{0 \vdash \text{::} t} \text{ (WAITING)} \qquad \frac{n \vdash t}{\vdash (n+1) \text{::} t} \text{ (FIRSTTOSS)} \\
 \\
 \frac{n \vdash t}{0 \vdash (n+1) \text{::} t} \text{ (CATCHTOSS)} \qquad \frac{n \vdash t}{n+1 \vdash 0 \text{::} t} \text{ (NOTOSS)} \\
 \\
 \frac{n \vdash t \quad m \vdash t \quad n \neq m}{n+1 \vdash (m+1) \text{::} t} \text{ (TOSSABLE)}
 \end{array}$$

**Fig. 4.** Validation rules for toss streams

## 2.2 Validity of Toss Streams

There are many invalid toss streams such as  $\langle 5.5.1 \rangle$  and  $\langle 2.4.0.1 \rangle$  shown in Fig. 3. These streams cannot be a juggling pattern because either more than one tossed object is simultaneously caught in a hand or a caught object disappears.

Formally, we say that a toss stream  $t$  is *invalid* if there are two distinct counts  $n$  and  $m$  such that the caught counts  $t_n + n$  and  $t_m + m$  coincide. If both  $t_n$  and  $t_m$  are more than 0, the coincidence implies that two objects are caught at the same time; if either  $t_n$  or  $t_m$  is 0, it implies that a caught object disappears. We say that  $t$  is *valid*, denoted by  $\models t$ , if  $t$  is not invalid.

Now we give an alternative definition for the validity of toss streams by coinduction, as shown in Fig. 4. We use two judgment forms,  $\vdash t$  and  $n \vdash t$ , with a toss stream  $t$  and non-negative integer  $n$ , for validation rules. A unary judgment  $\vdash t$  means that a toss stream  $t$  is valid and binary judgment  $n \vdash t$  means that it is valid even if an object in the air will be caught in  $n$  beats later. More specifically,  $0 \vdash t$  means that one object is being caught by a hand at the beginning of  $t$ . Therefore,  $n \vdash n \text{::} t$  must not hold so that any two objects are caught on a distinct beat (when  $n > 0$ ) and no objects disappear (when  $n = 0$ ). It is easy to check that  $n \vdash n \text{::} t$  has no proof for any  $n$  in our validation system.

The unary judgment has two rules: (WAITING) and (FIRSTTOSS). Rule (WAITING) implies that a 0-height toss on the starting beat is valid if the remaining toss stream is. Rule (FIRSTTOSS) implies that an  $(n+1)$ -height toss is valid if the remaining toss stream with an  $n$ -height object in the air is valid.

The binary judgment has three rules: (CATCHTOSS), (NOTOSS), and (TOSSABLE). Rule (CATCHTOSS) implies that an  $(n+1)$ -height toss on the current beat with an object caught at the moment is valid if the remaining toss stream with an  $n$ -height object in the air is valid. Rule (NOTOSS) implies that no toss on the current beat with an  $(n+1)$ -height object in the air is valid if the remaining toss stream with an  $n$ -height object in the air is valid. The height of the object decreases by one on each beat. Rule (TOSSABLE) implies that an  $(m+1)$ -height toss on the current beat with an  $(n+1)$ -height object in the air is

valid if the remaining toss stream is valid both with an  $n$ -height object and with an  $m$ -height object in the air where  $n \neq m$ . Note that there is no axiomatic rule like a ‘base case’ in the inductive definition. We define validity for toss streams in a coinductive manner because toss streams have a coinductive structure.

Our definition of validity is appropriate in the sense that a toss stream includes a collision if and only if it is not valid. This fact is formalized in the following theorem.

**Theorem 1 (soundness and completeness of validation rules).** *For any toss stream  $t$ , a judgment  $\vdash t$  holds if and only if  $t$  is valid, i.e.,  $\models t$ .*

The ‘only if’ statement establishes the soundness of the validation rules; that is, any toss stream derived from the validation rules must be valid. The ‘if’ statement establishes the completeness of the validation rules; that is, any valid toss stream can be derived from the validation rules. We will give a proof of this theorem in Section 2.5 after showing how our coinductive rules work for toss stream validation.

### 2.3 Validation of Toss Streams by Coinduction

We show how a given toss stream is validated in our validation system. The validity is proved by coinduction because the validation rules are coinductively defined.

We first give a brief review of *proof by coinduction* in a general setting. Coinduction is the dual of induction. While an assumption should have an inductive structure in an inductive proof, the conclusion should have a coinductive structure in a coinductive proof. To prove a coinductive statement  $P$  by coinduction, we can add  $P$  itself as an assumption, but it can be applied only in a restricted manner. Without this restriction, an arbitrary statement could be proved freely. We can understand the details of the restriction by recalling the *Curry-Howard correspondence* [How80].

The Curry-Howard correspondence is a connection between constructive logics and programming languages: there exists a constructive proof of a formula if and only if there exists a program of the corresponding type. For example, a formula  $A \rightarrow B \rightarrow A$  has a proof that  $A$  can be concluded from two assumptions  $A$  and  $B$  as  $a \rightarrow b \rightarrow a$  is a type of  $\lambda x : a. \lambda y : b. x$ . This is a natural correspondence between constructions of proofs and programs (in  $\lambda$ -terms).

The aforementioned restriction of coinductive proofs can be explained through the Curry-Howard correspondence, which connects them with co-recursive programs. A co-recursive program defines the greatest fixed point of an equation while a recursive program defines the least fixed point. Let us see a simple restriction for co-recursion in Coq, called a *guardedness condition*, which restricts occurrences of co-recursive calls. For example,

```
CoFixpoint toss_0 : toss := 0 ::~ toss_0.
CoFixpoint toss_51 : toss := 5 ::~ 1 ::~ toss_51.
```

are legally defined in Coq, but

```
CoFixpoint toss_loop : toss := toss_loop.
CoFixpoint toss_app : toss := 2 ::~ f (3 ::~ toss_app).
```

where  $f$  is a function over toss streams are not. This is because the first two equations have a trivial greatest fixed point while the latter two do not. Every co-recursive call should be guarded by only constructors in order to be syntactically judged a legal coinductive definition.

Coinductive proofs have the same restriction in terms of the Curry-Howard correspondence. As previously mentioned, in a coinductive proof of a statement  $P$ , we can add  $P$  itself as an assumption. The assumption cannot be directly used to conclude  $P$  as the definition of `toss_loop` should not be legal. In contrast, we can apply the added assumption for ‘guarded’ contexts in a sense. Since constructors correspond to judgment rules in a proof system through the Curry-Howard correspondence, the assumption can be used only after applying some rules. Coq provides the `cofix` tactic for adding the current goal to the assumptions, which is allowed only for restricted uses defined by the guardedness condition. Details on how the guardedness condition works are available elsewhere [Gim95, GC98].

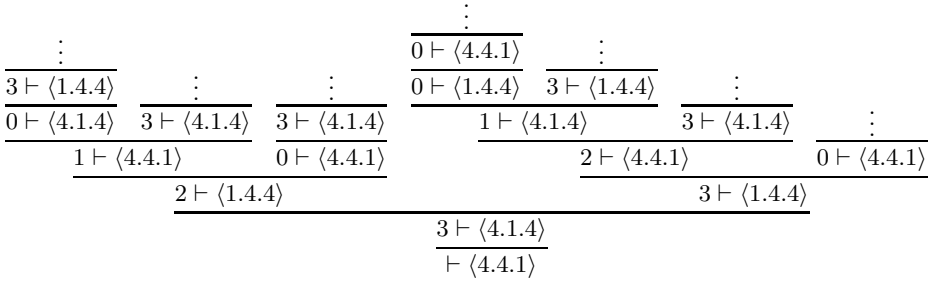
Now we prove the validity of toss patterns by coinduction. Consider the simple  $\langle 3.0.0 \rangle$  pattern (Fig. 1). The following proof tree illustrates the validity of the pattern.

$$\begin{array}{c}
 \vdots \\
 \hline
 0 \vdash \langle 3.0.0 \rangle \\
 \hline
 1 \vdash \langle 0.3.0 \rangle \\
 \hline
 2 \vdash \langle 0.0.3 \rangle \\
 \hline
 0 \vdash \langle 3.0.0 \rangle \\
 \hline
 1 \vdash \langle 0.3.0 \rangle \\
 \hline
 2 \vdash \langle 0.0.3 \rangle \\
 \hline
 \vdash \langle 3.0.0 \rangle
 \end{array}$$

The three dots at the top of the tree indicate that the tree has an infinite structure. In the coinductive proof, when deriving the judgment  $0 \vdash \langle 3.0.0 \rangle$  on the fourth line from the bottom, we put it as an assumption. We then use it to prove the same formula on the second line from the top<sup>2</sup>. The proof of the validity of  $\langle 3.0.0 \rangle$  in Coq is described as follows:

```
Theorem toss_300_valid : ⊢ toss_300.
Proof.
  assert (0 ⊢ toss_300); cofix;
  rewrite eq_unfold_toss; simpl; repeat constructor; auto.
Qed.
```

<sup>2</sup> We may choose  $2 \vdash \langle 0.0.3 \rangle$  or  $1 \vdash \langle 0.3.0 \rangle$  as a lemma instead of  $0 \vdash \langle 3.0.0 \rangle$ .



**Fig. 5.** Proof tree for validation of toss stream  $\langle 4.4.1 \rangle$  by mutual coinduction

where we first prove a lemma  $0 \vdash \langle 3.0.0 \rangle$  by coinduction using the `cofix` tactic. The successive instructions `rewrite eq_unfold_toss; simpl` unfold once the definition of the toss stream following Chlipala’s technique [Ch11]. Now what we have to prove is  $\vdash 3 :: 0 :: 0 :: \langle 3.0.0 \rangle$  and  $0 \vdash 3 :: 0 :: 0 :: \langle 3.0.0 \rangle$  under the assumption  $0 \vdash \langle 3.0.0 \rangle$ . Then the instruction `repeat constructor` applies judgment rules (without unfolding) as much as possible and `auto` proves trivial statements.

A proof tree for validation of the  $\langle 4.4.1 \rangle$  toss stream is more complex, as shown in Fig. 5, where there are many dots on the top. Every judgment beneath the dots occurs in a different branch of the tree, meaning that the validity is proved by mutual (or nested) coinduction. In the case of  $\langle 4.4.1 \rangle$ , we have to mutually prove many judgments that occur under the dots in the proof tree. Note that all of them can be derived from three judgments:  $0 \vdash \langle 4.4.1 \rangle$ ,  $1 \vdash \langle 4.4.1 \rangle$ , and  $2 \vdash \langle 4.4.1 \rangle$ . Hence we can prove the validity of  $\langle 4.4.1 \rangle$  in Coq as follows:

`Theorem toss_441_valid : ⊢ toss_441.`

`Proof.`

```

assert(0 ⊢ toss_441); cofix;
assert(1 ⊢ toss_441); cofix;
assert(2 ⊢ toss_441); cofix;
rewrite eq_unfold_toss; simpl; repeat constructor; auto.

```

`Qed.`

where we first prove three auxiliary lemmas,  $0 \vdash \langle 4.4.1 \rangle$ ,  $1 \vdash \langle 4.4.1 \rangle$ , and  $2 \vdash \langle 4.4.1 \rangle$ , by coinduction. The remaining instructions are similar to those for proving the validity of  $\vdash \langle 3.0.0 \rangle$ . The `auto` tactic is used for proving disequalities, which are caused by applying the (TOSSABLE) rule. The lemmas needed depend on the toss pattern to be validated, as discussed later.

It is easier to disprove the validity of toss streams than to prove their validity. Because of Theorem 1, it suffices to find two distinct  $n$  and  $m$  such that  $t_n + n = t_m + m$ . For example, a toss stream  $t$  represented by  $\langle 5.5.1 \rangle$  is invalid because both  $t_1 + 1$  and  $t_5 + 5$  are equal to 6. A toss stream  $t$  represented by  $\langle 2.4.0.1 \rangle$  is invalid because both  $t_0 + 0$  and  $t_2 + 2$  are equal to 2.

**Remark.** Our definition of validity is independent of the number of objects. It is possible to write a valid toss stream for an infinite number of objects, e.g., `toss_ints (= 0 ::~ 1 ::~ 2 ::~ 3 ::~ ...)` defined by

```
CoFixpoint toss_ints_from (n:nat) : toss :=
  n ::~ toss_ints_from (S n).
Definition toss_ints : toss := toss_ints_from 0.
```

which represents a toss stream  $t$  with  $t_n = n$ . Because  $t_n + n (= 2n)$  and  $t_m + m (= 2m)$  differ for distinct  $n$  and  $m$ , toss stream  $t$  is valid from the definition. Note that no object is caught by a hand on an odd beat. We have to manipulate an infinite number of objects since we have to toss a new object on every odd beat. This pattern was also considered by Buhler et al. [BEGW94]. From Theorem 1, the validity of the `toss_ints` pattern can be proved by coinduction in our validation system. Indeed, we can also directly illustrate the validity of the pattern by showing the lemma that  $n \vdash \text{toss\_int\_from } m$  holds when  $n < m$ , which can be proved by coinduction.

One may think that juggling patterns with infinite objects could not be valid. This is not a concern, however, if we consider only ‘periodic patterns’ represented by siteswap notation. Since no periodic pattern can generate a toss stream involving infinite objects [BEGW94], we use the definition above for the validity of toss streams.

## 2.4 General Tactic for Proving Validity of Patterns

As we mentioned above, we need to prepare auxiliary lemmas to prove the validity of a periodic toss stream. The lemmas needed depend on the toss pattern to be validated. For instance, the proof of the validity of  $\langle 1.2.3.4.5 \rangle$  is given by

```
Theorem toss_12345_valid : ⊢ toss_12345.
Proof.
  assert (0 ⊢ toss_12345); cofix;
  assert (2 ⊢ toss_12345); cofix;
  assert (4 ⊢ toss_12345); cofix;
  rewrite eq_unfold_toss; simpl; repeat constructor; auto.
Qed.
```

where we show three auxiliary lemmas  $0 \vdash \langle 1.2.3.4.5 \rangle$ ,  $2 \vdash \langle 1.2.3.4.5 \rangle$ , and  $4 \vdash \langle 1.2.3.4.5 \rangle$ , simultaneously. In general, to prove  $\vdash t$  in this way, the number of auxiliary lemmas of the form  $n \vdash t$  needed is the same as the number of orbits (i.e., the number of objects) that start being tossed  $n$  beats later from the initial position.

We shall present a tactic for proving the validity that works for arbitrary periodic toss streams as long as they are valid. This `certify_toss` tactic is defined by

```
Ltac certify_toss :=
  repeat (cofix; rewrite eq_unfold_toss; simpl;
    repeat constructor; auto; try simplify_eq).
```



**Table 1.** With-and-without comparison of `certify_toss` for valid patterns

siteswap	$\langle 1.2.3.4.5 \rangle$	$\langle 5.5.5.1 \rangle$	$\langle 7.5.3.1 \rangle$	$\langle 0.1.2.3.4.5.6 \rangle$
with tactic	1.071s / 0.549s	3.948s / 1.789s	4.814s / 2.108s	12.142s / 6.753s
without tactic	0.048s / 0.088s	0.080s / 0.379s	0.094s / 0.215s	0.107s / 0.933s

which simply repeatedly tries to prove any statement by coinduction. The `auto` tactic is employed to use an assumption and solve the trivial disequality. It finally tries the `simplify_eq` tactic to solve the remaining disequalities because the `auto` tactic cannot prove disequalities between large numbers, on which we will give an observation later.

The tactic will end when all statements are proved by applying judgment rules and assumptions. Informally, termination of the tactic can be shown as follows. The tactic contains two `repeat` tactics that may cause non-termination. The inner `repeat` obviously terminates because the `constructor` tactic tries to apply a judgment rule without unfolding the toss stream. Let us show termination of the outer `repeat`. The only tactic that forks goals is `constructor`. The number of possible goals added by the `constructor` tactic is finite since each added goal has the form  $n \vdash t$ , where the number of possible  $n$ 's and  $t$ 's is bounded. This is because the toss stream is periodic and the number  $n$  must be less than the number in the toss stream. Furthermore, the same goal cannot be tried to prove twice. Once the goal is added by `constructor`, it is pushed as an assumption by the `cofix` tactic. After the goal is added at the next time, the `auto` tactic must have solved it from the assumption. As a consequence, all of the goals of the form  $n \vdash t$  are solved. Since the `auto` tactic which may use the assumption added by `cofix` is applied after the `constructor` tactic, the guardedness condition for coinduction cannot be violated. If the `certify_toss` tactic is applied to an invalid toss stream, Coq fails to prove it, leaving many incorrect goals like  $n \neq n$ , which are wrecks of tried application of the (TOSSABLE) rule.

Using the `certify_toss` tactic has an efficiency problem, however, because it tries to prove all occurred judgments by coinduction. This is much less efficient than a proof using the necessary lemmas as done in Section 2.3.

Table 1 shows a with-and-without comparison of the `certify_toss` tactic for four valid siteswap patterns. Each cell in the table has the form  $t_{\text{sol}}/t_{\text{con}}$  where  $t_{\text{sol}}$  and  $t_{\text{con}}$  are the times for goal solving phase and proof term construction phase, respectively, and both are the total CPU time measured with the `Time` command in Coq 8.3pl2 running on a MacBook Air computer (1.8 GHz Intel Core i7 and 4 GB memory). The time for proof term construction includes guardedness checking. The siteswaps,  $\langle 1.2.3.4.5 \rangle$ ,  $\langle 5.5.5.1 \rangle$ ,  $\langle 7.5.3.1 \rangle$ , and  $\langle 0.1.2.3.4.5.6 \rangle$ , denote the toss patterns in which 3, 4, 4, and 3 objects are manipulated, respectively. We can see that proving the validity with the tactic is much less efficient than without it particularly not only when the number of objects is large but also when the siteswap is long. This is because the tactic generates more goals to be proved for longer siteswaps. To improve the performance, we need to modify

the definition of the tactic so that unnecessary statements are not tried to prove, which is difficult to do in general.

The tactic-based approach has other problems even if we cleverly redefine the tactic. First, we cannot guarantee termination of the tactic in Coq. We need to certify outside Coq that the body of the `repeat` tactic eventually fails and that all recursively defined tactics terminate. Second, we cannot guarantee the correctness of the tactics in Coq. We need to certify that no tactic leaves unexpected goals after its application. In fact, one may implement the `certify_toss` tactic without `simplify_eq` at the end because it happens to succeed in proving all examples in the benchmark above. However, without `simplify_eq`, it fails to prove the validity of  $\langle 9, 1 \rangle$ , which is a valid siteswap pattern. This wrong tactic leaves obvious disequalities,  $6 \neq 8$ , that fails to be proved by the `auto` tactic because of the limitation of the search depth. To solve this problem, we should either modify the definition of `certify_toss` as above or register the `discriminate` tactic as a hint to make `auto` more powerful.

These problems with the tactic-based approach can be solved by using a validity checking algorithm for arbitrary periodic toss patterns in Section 3.

## 2.5 Proof of Theorem 1

Theorem 1 is illustrated by independently proving the ‘only if’ statement and the ‘if’ statement. Both statements are proved in Coq. For convenience, we extend the definition of  $\models$  to a binary relation: for a non-negative integer  $n$  and a toss stream  $t$ , we say  $n \models t$  if both  $\vdash t$  holds and  $n \neq t_m + m$  for any non-negative integer  $m$ .

*Proof of the only-if statement.* The statement establishes the soundness of our validation rules; that is,  $\vdash t$  implies  $\models t$  for any toss stream  $t$ . This can be shown by using two lemmas:

- (i)  $x \vdash t$  implies  $\models t$  for any non-negative integer  $x$ , and
- (ii)  $x \vdash t$  implies  $x \neq t_m + m$  for any non-negative integer  $x$ .

Recall that the definition of  $\models t$  is that, for any  $m$  and  $n$  with  $m \neq n$ ,  $t_m + m \neq t_n + n$  holds. Hence, we can use induction on these  $m$  and  $n$  to prove a statement in the form  $\models t$ . Statement (ii) is proved by induction on either  $m$  or  $n$ . To prove statement (i), it suffices to show that  $x \vdash t$  implies  $x \neq t_m + m$  for any  $m$  because of statement (ii) and the definition of  $x \models t$ , which is proved by induction on  $x$ . The ‘only if’ statement above is proved using statement (i) by induction on either  $m$  or  $n$ , again.

*Proof of the if statement.* The statement establishes the completeness of our validation rules; that is,  $\models t$  implies  $\vdash t$  for any toss stream  $t$ . This can be shown by using a lemma:

- (iii)  $x \models t$  implies  $x \vdash t$  for any non-negative integer  $x$ ,

which is proved by coinduction with case analyses on  $x$  and  $t$ . Then the if-statement above is proved by coinduction, again.

**Input:** a toss pattern as a list  $l = [a_1, \dots, a_p]$

**Output:** whether the toss stream  $t = \langle a_1, \dots, a_p \rangle$  is valid

1. Add a number  $n$  to the  $n$ -th element in  $l$  for each  $n$ .
2. Take modulo  $p$  for each element.
3. If the obtained list contains no pair of elements with the same numbers,  $t$  is valid; otherwise not.

**Fig. 6.** Algorithm for validating toss streams represented by siteswap

### 3 Validity Checking Algorithm for Periodic Toss Patterns

To avoid the problems related to user-defined tactics mentioned in Section 2.4, we implement a general method for toss pattern validation. We use the algorithm developed by Buhler et al. [BEGW94] to validate periodic toss streams. They proved the correctness of the algorithm in their formulation, in which each juggling pattern was given by a bijective function over integers. In this section, we show the soundness and completeness of the algorithm in our formulation, in which each juggling pattern is represented by a coinductive data structure.

Let us formalize a periodic toss pattern as a nonempty list in Coq:

```
Definition toss_pattern := { ns:list nat | nil <> ns }.
```

where  $\{ x:A \mid P x \}$  represents a sigma type of  $A$  satisfying the  $P$  property. We use the `pattern_to_toss` function which takes a toss pattern and returns a periodic toss stream obtained by repeating the pattern. The function is defined by a co-recursion:

```
CoFixpoint pattern_to_toss (pat:toss_pattern) : toss :=
  head pat ::~ pattern_to_toss (rotate pat).
```

where `head pat` returns the first element of `pat` (this will not fail because `pat` is not empty) and `rotate pat` returns a pattern obtained by removing the first element and appending it to the end of the pattern.

#### 3.1 Siteswap Verification Algorithm

Figure 6 shows the algorithm used to determine the validity of a toss stream given by siteswap notation. It is easy to define the procedure in Coq:

```
Definition passed_toss_pattern (pat:toss_pattern) : Prop :=
  NoDup (map (modulo_length pat) (add_offset pat)).
```

where `modulo_length pat` is a function computing the remainder of division of a given number by the length of the toss pattern `pat`, and `add_offset pat` returns a list of the same length as the toss pattern `pat` the elements of which are incremented by the offset position from the head of the list. The `map` function is the same as in functional programming languages, and `NoDup` checks whether

a given list has no duplicated elements. This procedure can be implemented as an algorithm due to the decidability of `pass_toss_pattern`, which obviously holds because the `NoDup` property is decidable for lists of natural numbers. The `passed_toss_pattern` property is useful for giving a precise definition of valid and periodic patterns if we need:

```

Definition valid_toss_pattern :=
  { pat:toss_pattern | passed_toss_pattern pat }.
    
```

We prove the soundness and completeness of the validation algorithm with the following theorem:

**Theorem 2 (soundness and completeness of validation algorithm).** *Let  $t$  be a toss stream generated by a toss pattern  $p$ , i.e.,  $t = \text{pattern\_to\_toss } p$ . Then  $\vdash t$  if and only if the algorithm outputs yes for  $p$ .*

The ‘if’ statement of the theorem represents soundness while the ‘only if’ statement represents completeness. Note that this theorem establishes correctness of the algorithm only for a toss stream generated by repeating a toss pattern. It excludes non-periodic patterns like `toss_ints` mentioned in the previous section. There is no general scheme to determine the validity of non-periodic patterns.

The proof of this theorem in Coq is done in a way similar to the original one by Buhler et al. [BEGW94]. Because of Theorem 1, it suffices to show that the algorithm exactly determines the validity of toss streams in terms of  $\models$ . We could hence prove it in Coq without coinduction as Buhler et al. did. It would be interesting to consider a *direct* coinductive proof of the theorem without relying on Theorem 1.

## 4 Conclusion

We have formalized the validation of juggling toss patterns, called siteswap, in terms of coinduction. This is a natural application of coinduction since the validity involves an infinite phenomenon. The formalization was implemented in Coq. We have introduced a general tactic that proving the validity of toss streams using mutual coinduction. Furthermore, we have implemented the algorithm developed by Buhler et al. for checking the validity of siteswap and proved its correctness.

We could formalize the validation of toss patterns by using a list of non-conflict orbits. Toss patterns could be defined by using mixture of induction and coinduction: a toss pattern could be defined as a list of orbits that is an inductive structure; the orbits and their non-conflicting property could be defined by coinduction. This formalization may be useful for showing coincidence between the number of orbits (that is, the length of the orbit list) and the average of the numbers in the siteswap pattern [BEGW94]. This coincidence indicates that the number of objects is finite if the toss pattern is periodic. Note that the converse is not true. We can construct a non-periodic toss pattern by replacing binary

numbers in the Thue-Morse sequence [OEIS] with two patterns (2) and (3.1), i.e.,  $2 \dashv\vdash 3 \dashv\vdash 1 \dashv\vdash 3 \dashv\vdash 1 \dashv\vdash 2 \dashv\vdash 3 \dashv\vdash 1 \dashv\vdash 2 \dashv\vdash 3 \dashv\vdash 1 \dashv\vdash \dots$ , which is a toss pattern in which two objects are manipulated.

**Acknowledgments.** The author is grateful to Sebastian Fischer for inspiring me to think about a tactic for validating arbitrary toss patterns. He also thanks to the anonymous reviewers for their helpful comments and suggestions.

## References

- [BEGW94] Buhler, J., Eisenbud, D., Graham, R., Wright, C.: Juggling Drops and Descents. *American Mathematical Monthly* 101(6), 507–519 (1994)
- [Chl11] Chlipala, A.: *Certified Programming with Dependent Types*. MIT Press (2011), <http://adam.chlipala.net/cpdt/>
- [GC98] Giménez, E., Castéran, P.: A Tutorial on [Co-]Inductive Types in Coq (1998)
- [Gim95] Giménez, E.: Codifying Guarded Definitions with Recursive Schemes. In: Smith, J., Dybjer, P., Nordström, B. (eds.) *TYPES 1994*. LNCS, vol. 996, pp. 39–59. Springer, Heidelberg (1995)
- [How80] Howard, W.A.: The Formulae-As-Types Notion of Construction. In: Seldin, J.P., Hindley, J.R. (eds.) *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490. Academic Press (1980)
- [LG09] Leroy, X., Grall, H.: Coinductive big-step operational semantics. *Information and Computation* 207(2), 284–304 (2009)
- [NU09] Nakata, K., Uustalu, T.: Trace-Based Coinductive Operational Semantics for While. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 375–390. Springer, Heidelberg (2009)
- [OEIS] OEIS Foundation Inc. Sequence A010060. The On-Line Encyclopedia of Integer Sequences, published electronically at <http://oeis.org/A010060>
- [San09] Sangiorgi, D.: On the Origins of Bisimulation and Coinduction. *ACM Transactions on Programming Languages and Systems* 31(4), 15:1–15:41 (2009)

# Proof Pearl: Abella Formalization of $\lambda$ -Calculus Cube Property

Beniamino Accattoli

INRIA and LIX (École Polytechnique) - Palaiseau, France  
Carnegie Mellon University - Pittsburgh, PA, USA

**Abstract.** In 1994 Gerard Huet formalized in Coq the cube property of  $\lambda$ -calculus residuals. His development is based on a clever idea, a beautiful inductive definition of residuals. However, in his formalization there is a lot of noise concerning the representation of terms with binders. We re-interpret his work in Abella, a recent proof assistant based on higher-order abstract syntax and provided with a nominal quantifier. By revisiting Huet’s approach and exploiting the features of Abella, we get a strikingly compact and natural development, which makes Huet’s idea really shine.

## 1 Introduction

The confluence or Church-Rosser theorem of  $\lambda$ -calculus has been formalized in several proof assistants, and it is probably the theorem with the highest number of formalized proofs [26,31,21,32,27,30,17,5,16]. In [17] Huet formalizes in Coq also a deeper result, the cube property of  $\lambda$ -calculus residuals (due to Jean-Jacques Lévy [20,4]). This paper presents a new, simple formalization of this result, developed in Abella [9,8], a recent proof-assistant based on higher-order abstract syntax (HOAS) [7,23] and provided with a nominal quantifier [13,12,11,24,10,2].

Residual systems are a standard tool in rewriting [20,18,19,22,15,3,34]. In particular, they are at the basis of the advanced rewriting theory of  $\lambda$ -calculus and orthogonal rewriting systems (standardization, neededness, Lévy’s families and optimality, inside-out reductions, see [20,18,19,3,34]). Roughly, one first introduces a mechanism to track redexes along reductions (typically using positions or underlinings), so that it is possible to say which are the residuals of a redex  $r$  after another redex. Then, one shows that any given span  $u_2 \leftarrow t \rightarrow u_1$  can be closed by simply reducing on both sides the residuals of one redex after the other. The idea is that residuals refine confluence providing a *minimal* closure of confluence diagrams. The abstract theory of residual systems—which is independent from  $\lambda$ -calculus—is based on three axioms, the cube property plus two other minor axioms (see [34], Chapter 8.7). The development in this paper essentially proves that  $\lambda$ -calculus admits a residual system, but we will not enter into the details of the abstract theory.

A delicate point is how to define residuals for a given calculus. In [17] Huet presents an elegant and compact solution for  $\lambda$ -calculus: he uses a simple ternary relation over terms with underlinings, defined by induction on the structure of the first term. However, Huet represents (marked)  $\lambda$ -terms using de Bruijn indexes, and a relevant part of his development deals with the properties of indexes, substitution and lifting.

Initially, we repeated Huet’s development to see how much Abella—being a proof assistant based on HOAS—could help in simplifying Huet’s work. All the troubles about indexes, lifting and substitution disappear, this was expected. However, along the way we realized that other simplifications were possible (the first two are independent from Abella):

1. *Marks*: Huet underlines applications, which requires to introduce a notion of well-formed term (*regular terms* in [17]) and to show that various operations preserve well-formedness. According to common practice we rather mark redexes (as in [3], for instance), so that any marked term is well-formed, and some lemmas disappear.
2. *Rewriting*: by analyzing inductions and dependencies between lemmas we simplify the statements and the number of lemmas required to prove the cube property. In particular, Huet recognizes the so-called *prism property* as more fundamental than the cube property, but we show that the direct proof of the cube property is not harder than the proof of the prism property. This also agrees with the clean abstract theory in [34] (which did not exist at the time of [17]), where there are examples of residual systems enjoying the cube property but not the prism property. Actually, we show that for  $\lambda$ -calculus it is possible to prove both properties with the same induction.
3. *Contexts*: in HOAS-based proof assistants  $\alpha$ -equivalence and substitution are primitive notions, but induction usually requires to consider predicates inside contexts of local assumptions (called *worlds* in Twelf [28], *schemas* in Beluga [29,6]) and prove properties about them. With respect to the untyped  $\lambda$ -calculus these contexts are artifacts, since they do not belong to the informal theory. The nominal quantifier  $\nabla$  (nabla) of Abella, not available in other HOAS settings, allows to formalize the untyped  $\lambda$ -calculus circumventing the use of contexts.

The final result is quite striking: we formalize a property subsuming both the cube and prism properties using only two definitions and one auxiliary lemma. Moreover, the development follows exactly the informal, pen-and-paper reasoning: there is no need to care about indexes,  $\alpha$ -equivalence, substitution or contexts.

The next section contains an introduction to residuals and the way Huet represents them. In Section 3 we present the formal development, also explaining the representation of  $\lambda$ -terms. Section 4 discusses some variations over our development.

The sources of the development can be found on-line [1].

## 2 The Diamond and Cube Properties, Informally

*The diamond property.* A rewriting system  $(S, \rightarrow)$  is **confluent** when for any  $s \in S$ :

$$s_2 \xleftarrow{*} s \rightarrow^* s_1 \text{ implies } \exists t \text{ s.t. } s_1 \rightarrow^* t \xleftarrow{*} s_2$$

The corresponding diagram is in Fig. 1a (solid arrows denote the reductions in the hypothesis, dashed arrows denote the reductions in the conclusion). A stronger notion is the **diamond property** (Fig. 1b):

$$s_2 \leftarrow s \rightarrow s_1 \text{ implies } \exists t \text{ s.t. } s_1 \rightarrow t \leftarrow s_2$$

The diamond property implies confluence, but not the converse. Unfortunately,  $\beta$ -reduction does not enjoy the diamond property (because of duplications/erasures). The standard technique (due to Tait and Martin-Löf) for proving confluence of  $\beta$ -reduction is to use a parallel reduction. The idea is to extend  $\beta$ -reduction to a reduction  $\Rightarrow$  so that:

1.  $\Rightarrow$  enjoys the diamond property, and thus confluence;
2. confluence of  $\Rightarrow$  implies confluence of  $\rightarrow_\beta$ .

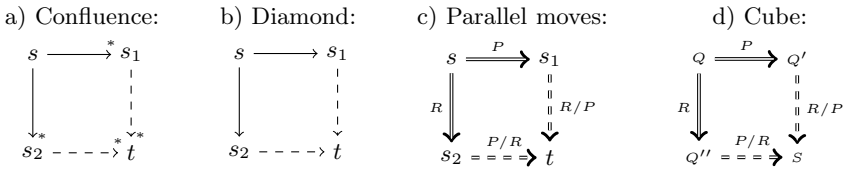


Fig. 1. Diagrams

The parallel reduction  $\Rightarrow$  is defined as follows<sup>1</sup>:

$$\frac{}{x \Rightarrow x} \Rightarrow\text{-var} \qquad \frac{t \Rightarrow t'}{\lambda x.t \Rightarrow \lambda x.t'} \Rightarrow\text{-}\lambda$$

$$\frac{t \Rightarrow t' \quad u \Rightarrow u'}{t u \Rightarrow t' u'} \Rightarrow\text{-}@ \qquad \frac{t \Rightarrow t' \quad u \Rightarrow u'}{(\lambda x.t) u \Rightarrow t' \{x/u'\}} \Rightarrow\text{-}\beta$$

The proof of confluence for  $\rightarrow_\beta$  is in two parts. The first part is to prove that  $\Rightarrow$  has the diamond property. The second part is to deduce confluence of  $\rightarrow_\beta$

<sup>1</sup> There is a subtlety: sometimes (in [34] for instance) *parallel reduction* denotes the reduction which reduces in parallel disjoint redexes only, while the reduction presented here (which may reduce nested redexes using rule  $\Rightarrow\text{-}\beta$ , for instance  $(\lambda x.x)((\lambda y.y)z) \Rightarrow z$ ) is called *simultaneous* or *multi-step* reduction. Often, however, the two concepts are not distinguished and the simultaneous reduction is called *parallel* (in [33] for instance). We follow this second tradition.



from the diamond property for  $\Rightarrow$ . Clearly, one has  $\rightarrow_{\beta} \subseteq \Rightarrow \subseteq \rightarrow_{\beta}^*$ , which implies  $\rightarrow_{\beta}^* \subseteq \Rightarrow^* \subseteq \rightarrow_{\beta}^*$ , *i.e.*  $\rightarrow_{\beta}^* = \Rightarrow^*$ . A straightforward induction shows that the diamond property of  $\Rightarrow$  implies the diamond property of  $\Rightarrow^*$ , which is nothing but confluence of  $\rightarrow_{\beta}$ .

The first part of the confluence proof uses structural induction over terms with binders, and it is related to the cube property for residuals, so we shall focus on it. The second part is based on the so-called *strip lemma*, it mostly involves first-order reasoning, and its formalization in Abella does not differ much from the other developments of confluence in the literature (for instance [17,27]), and thus it will be omitted<sup>2</sup>.

The proof of the diamond property requires a substitution lemma:

**Lemma 1 (substitutivity of  $\Rightarrow$ ).** *If  $t \Rightarrow t'$  and  $u \Rightarrow u'$  then  $t\{x/u\} \Rightarrow t'\{x/u'\}$ .*

*Proof.* By induction on  $t \Rightarrow t'$ . Two base cases (rule  $\Rightarrow$ -var):  $x \Rightarrow x$ , which gives  $x\{x/u\} = u \Rightarrow u' = x\{x/u'\}$ , and  $y \Rightarrow y$  (with  $y \neq x$ ) for which  $y\{x/u\} = y \Rightarrow y = y\{x/u'\}$ . The cases  $\Rightarrow$ - $\lambda$  and  $\Rightarrow$ -@ follow immediately from the *i.h.*. The case  $\Rightarrow$ - $\beta$ : if  $t = (\lambda y.s)v$  and  $t' = s'\{y/v'\}$  then by *i.h.*  $s\{x/u\} \Rightarrow s'\{x/u'\}$  and  $v\{x/u\} \Rightarrow v'\{x/u'\}$ , then by  $\Rightarrow$ - $\beta$  we get  $r = (\lambda y.s\{x/u\})v\{x/u\} \Rightarrow s'\{x/u'\}\{y/v'\{x/u'\}\} = r'$ . We conclude, since  $r = t\{x/u\}$  and  $r' = t'\{x/u'\}$ .

Then (both proofs are formalized in Section 3, Figure 5, page 182):

**Theorem 1 (diamond property of  $\Rightarrow$ ).**  *$s_2 \Leftarrow s \Rightarrow s_1$  implies  $\exists t$  s.t.  $s_1 \Rightarrow t \Leftarrow s_2$ .*

*Proof.* By induction on  $s \Rightarrow s_1$  and case analysis of  $s \Rightarrow s_2$ , using Lemma 1. If  $s = x \Rightarrow x = s_1$  then  $s_2$  can only be  $x$  and there is nothing to prove. The  $\Rightarrow$ - $\lambda$  case follows by the *i.h.*. Both  $\Rightarrow$ -@ and  $\Rightarrow$ - $\beta$  cases have two subcases, corresponding to  $s \Rightarrow s_2$  being  $\Rightarrow$ -@ or  $\Rightarrow$ - $\beta$ . In every subcase one has to use the *i.h.*, and whenever one of the hypothesis is  $\Rightarrow$ - $\beta$  it is necessary to apply Lemma 1.

*Residuals.* The diamond property of  $\Rightarrow$  can be strengthened with information about which redexes are reduced. First, one needs to introduce a mechanism for tracing redexes through reductions.

Let us stress that we need to trace *sets* of redexes. Indeed, consider the following reductions, where  $I = \lambda z.z$ :

$$(\lambda x.xx) (II) \rightarrow_{\beta} (II)(II) \quad (\lambda x.y) (II) \rightarrow_{\beta} y$$

The redex  $II$  may be duplicated, getting two residuals in the reduct, or erased, having no residual. By the way, this is also the reason why  $\beta$ -reduction does not enjoy the diamond property.

<sup>2</sup> The second part has nonetheless been formalized in Abella, it can be found in [1].

Consider a term  $s$  and two sets of redexes  $R$  and  $P$  in  $s$ . If  $s \Rightarrow v$  by reducing the redexes in  $P$  then there must be a way of describing what is left in  $v$  of the redexes in  $R$  after the reduction of  $P$ , *i.e.* of describing *the set of residuals of  $R$  after  $P$* , noted  $R/P$ .

Now, assume to know how to define and trace residuals, and to have a refined notion of reduction  $s \xrightarrow{R} v$ , which reduces the set  $R$  of redexes in  $s$ . The diamond property enriched with residuals—usually called the *parallel moves* property—is in Fig. 1c. The refinement essentially says that residuals allow to close the diagram in a *minimal* way<sup>3</sup>.

The point is now how to define residuals and their reduction. Huet uses a brilliant method, but unfortunately in [17] his idea does not shine as it could, because of too many technical details. One of the aims of this paper is to bring to the fore the elegance of Huet’s approach.

First of all, one needs to represent sets of redexes. This can easily be done introducing a new constructor  $(\lambda x.s)v$  for marked redexes and say that a set of redexes  $R$  in a term  $t$  is the term with marks  $T$  obtained from  $t$  by marking the redexes in  $R$ . So we switch to the following grammar of marked terms:

$$R ::= x \mid \lambda x.R \mid RR \mid \underline{(\lambda x.R)R}$$

For instance the four possible sets of redexes of  $(\lambda x.(II))I$  are:

$$(\lambda x.(II))I \quad (\lambda x.(\underline{II}))I \quad \underline{(\lambda x.(II))I} \quad \underline{(\lambda x.(\underline{II}))I}$$

In [17] the marks are on applications, which may not be redexes. Marking applications requires a notion of *well-marked term* (*regular terms* in [17]) which comes with various annoying complications. The choice of marking redexes is our first simplification (which is standard, we are not claiming originality).

Huet’s contribution is the definition of residuals. For a marked redex  $R$  let its **support** be the  $\lambda$ -term obtained by removing all underlinings. Given a term  $t$  we want to define  $R/P$ , the residuals of the redexes in the set  $R$  after the reduction of another set  $P$  of redexes of  $t$ . Note that both  $R$  and  $P$  are terms, and that they share the same support  $t$ . A first step towards the definition of  $R/P$  is to forget  $t$  and consider  $R$ , which is  $t$  plus the information about the redexes we want to track. The idea is to see  $R/P$  simply as the target  $R'$  of a reduction step  $R \xrightarrow{P} R'$ . Of course, now the point is how to define  $R \xrightarrow{P} R'$ . What is particularly nice is that it can be defined by a simple structural induction over  $R$ , exploiting the fact that  $R$  and  $P$  have the same support:

---

<sup>3</sup> This minimality can be stated mathematically as the existence of pushouts in a certain category of reductions sequences, but its precise formulation requires to introduce permutation equivalence, which is beyond the scope of this paper.

$$\begin{array}{c}
 \frac{}{x \xrightarrow{x} x} \\
 \\
 \frac{R \xrightarrow{P} R' \quad S \xrightarrow{Q} S'}{RS \xrightarrow{PQ} R'S'} \qquad \frac{R \xrightarrow{P} R' \quad S \xrightarrow{Q} S'}{(\lambda x.R)S \xrightarrow{(\lambda x.P)Q} (\lambda x.R')S'} \\
 \\
 \frac{R \xrightarrow{P} R' \quad S \xrightarrow{Q} S'}{(\lambda x.R)S \xrightarrow{(\lambda x.P)Q} R'\{x/S'\}} \qquad \frac{R \xrightarrow{P} R' \quad S \xrightarrow{Q} S'}{(\lambda x.R)S \xrightarrow{(\lambda x.P)Q} R'\{x/S'\}}
 \end{array}$$

An example: if  $R = (\lambda x.xx)$  (II) and  $P = (\lambda x.xx)$  (II) then  $R/P$  is the marked term  $R' = (\underline{II})(\underline{II})$ , because one easily derives:

$$(\lambda x.xx) (\underline{II}) \xrightarrow{(\lambda x.xx) (\underline{II})} (\underline{II})(\underline{II})$$

Now, we have all the ingredients to prove the parallel moves property. However, a stronger property—the *cube* property, due to Jean-Jacques Lévy [20,4]—can now be expressed. Note that in Fig. 1.c the starting term is a  $\lambda$ -term  $s$ . Observe that any  $\lambda$ -term  $s$  is a marked term: it represents the empty set of redexes of  $s$ . By simply replacing  $s$  with a generic set of redexes, *i.e.* a marked term  $Q$ , we get the cube property (see Fig. 1.d). The cube enriches the parallel moves property with a sort of *contextual coherence*: the two sides of the diagram give the same term *and act in the same way on any other set of redexes in the starting term*.

By repeating Huet’s development in Abella and then analyzing the structure of the formal proof we realized that the cube property can be proved exactly as the diamond property of  $\Rightarrow$ . One needs to first prove the following lemma (called *commutation lemma* in [17]):

**Lemma 2 (substitutivity of  $\xrightarrow{P}$ ).** *If  $R \xrightarrow{P} R'$  and  $S \xrightarrow{Q} S'$  then  $R\{x/S\} \xrightarrow{P\{x/Q\}} R'\{x/S'\}$ .*

The proof is a simple induction on  $R \xrightarrow{P} R'$  (analogously to Lemma 1). Then one gets 1:

**Theorem 2 (cube property of  $\xrightarrow{P}$ ).**  *$Q' \xleftarrow{P} Q \xrightarrow{R} Q''$  implies  $\exists S$  s.t.  $Q' \xrightarrow{R/P} S \xleftarrow{P/R} Q''$ .*

The proof is by induction on  $Q \xrightarrow{P} Q'$  and case analysis of  $Q \xrightarrow{R} Q''$ , using Lemma 2 (their formalization is in the last page, after the bibliography).

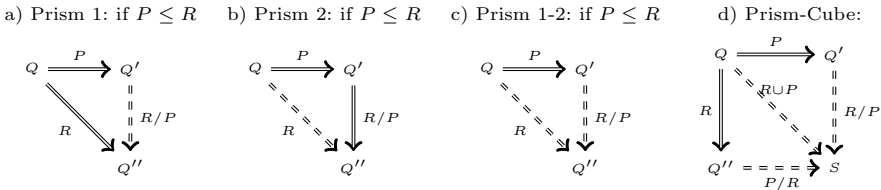
*The prism property.* In [17] Huet argues that the cube property follows from a more primitive property, the prism property. We need a definition: given two

---

<sup>4</sup> The more accurate but less readable statement is:  $Q' \xleftarrow{P} Q \xrightarrow{R} Q''$  implies  $\exists S, R', P'$  s.t.  $P \xrightarrow{R} P', R \xrightarrow{P} R',$  and  $Q' \xrightarrow{R'} S \xleftarrow{P'} Q''$ .

$\frac{}{x \leq x}$	$\frac{R \leq R'}{\lambda x. R \leq \lambda x. R'}$	$\frac{R \leq R' \quad S \leq S'}{RS \leq R'S'}$
	$\frac{R \leq R' \quad S \leq S'}{(\lambda x. R)S \leq (\lambda x. R')S'}$	$\frac{R \leq R' \quad S \leq S'}{(\lambda x. R)S \leq (\lambda x. R')S'}$
$\frac{}{x \cup x = x}$	$\frac{R \cup P = Q \quad S \cup T = U}{R \cup S \cup P \cup T = Q \cup U}$	$\frac{R \cup P = Q \quad S \cup T = U}{(\lambda x. R)S \cup (\lambda x. P)T = (\lambda x. Q)U}$
$\frac{R \cup P = Q}{\lambda x. R \cup \lambda x. P = \lambda x. Q}$	$\frac{R \cup P = Q \quad S \cup T = U}{(\lambda x. R)S \cup (\lambda x. P)T = (\lambda x. Q)U}$	$\frac{R \cup P = Q \quad S \cup T = U}{(\lambda x. R)S \cup (\lambda x. P)T = (\lambda x. Q)U}$

**Fig. 2.** The order ( $\leq$ ) and the union operation ( $\cup$ ) for sets of redexes/marked terms



**Fig. 3.** Prism diagrams

marked terms  $R$  and  $S$  with the same support, let  $R \leq S$  hold if  $S$  has all the marks in  $R$  and possibly more (see Fig. 2 for an inductive definition). Then the **prism property** is given by the two implications in Fig. 3a-b. The idea is that the prism gives one half of the cube property, which then follows by a symmetry argument (actually one needs only Fig. 3a).

By looking closely at the formalized proofs we realized that both parts of the prism property can be proved by induction on  $P \leq R$ . So that it should rather be stated as in Fig. 3c. The point is that the cube property essentially follows from the same induction.

Let us clarify this point. Given marked redexes  $R$  and  $S$  sharing the same support, let  $R \cup S$  be the marked term with all the marks of  $R$  and all the marks of  $S$  (see Fig. 2). If  $R \leq S$  then there is  $P$  s.t.  $R \cup P = S$ . So doing induction on  $R \leq S$  is essentially equivalent to inducting on  $R \cup P$ . The cube property in Fig. 1d can be proved by induction on  $R \cup P$ . Therefore, the same induction also proves the diagram in Fig. 3d, which puts together the prism and the cube property. Last, induction on  $R \cup P$  can be replaced by induction over  $Q \xrightarrow{P} Q'$  and case analysis of  $Q \xrightarrow{R} Q''$ . We then get:

**Theorem 3 (Prism-cube property of  $\xrightarrow{P}$ ).**  $Q' \xleftarrow{P} Q \xrightarrow{R} Q''$  implies  $\exists S$  s.t.  $Q' \xrightarrow{R/P} S \xleftarrow{P/R} Q''$  and  $Q \xrightarrow{R \cup P} S$ .

In the theory of *abstract residuals systems* [34] (Chapter 8.7) the cube property is one of the axioms while the prism property (there called *triangle property*) is not required to hold, and in fact there are examples of residual systems where it fails ([34], 8.7.29, page 440, or the rewriting system obtained orienting the associativity rule, see also [22]).

As for the cube property, it is possible to refine the diamond property into the *prism-diamond property*: if  $s_2 \Leftarrow s \Rightarrow s_1$  then  $\exists t$  s.t.  $s_1 \Rightarrow t \Leftarrow s_2$  and also  $s \Rightarrow t$ . Actually, it is this enriched property—proved exactly as the diamond property—that we have formalized in Abella.

### 3 The Diamond and Cube Properties, Formally in Abella

*Abella*. Abella is an interactive theorem prover developed by Andrew Gacek [9,8], and based on the logic  $G$  developed by Gacek, Miller, and Nadathur [13,12,11]. Abella uses the higher-order abstract syntax (HOAS) approach to binders [7,23] and it is provided with a nominal quantifier  $\nabla$  (nabla) [13,12,11,24,10,2] (which has a subtle proof theory that shall not be treated here). Being based on HOAS, Abella provides a primitive handling of  $\alpha$ -equivalence and capture-avoiding substitution. By exploiting  $\nabla$ , Abella is also able to mix inductive and co-inductive reasoning.

Many domains in which Abella has been used involve reasoning about typing judgements. In order to facilitate the treatment of such judgements, a *second logic* is defined within Abella. This second logic, a small intuitionistic *specification logic*, can be used to directly treat a range of typing judgments; this is what is sometimes called the *two-levels approach* [14]. A key property is that the specification logic satisfies cut-elimination. This fact allows to use cut-elimination as a tactic, and derive substitution lemmas *for free*. For instance, subject-reduction proofs can often be written very cleanly.

If one uses Abella to reason on specifications that do not involve typing judgements, then the built-in specification logic is not needed. For example, many properties of the untyped  $\pi$ -calculus have been proved using  $\nabla$ , induction, and co-induction, without using the two-level logic architecture of Abella [25]. Since our subject here is the untyped  $\lambda$ -calculus, we shall similarly find no need for Abella’s second level of logic.

*$\lambda$ -terms*. The encoding of  $\lambda$ -terms we use is standard (for HOAS). The encoding of marked terms extends the encoding of  $\lambda$ -terms (whose constructors are now renamed **mabs** and **mapp**) with **mredex**  $R\ S$ , which represents the marked redex  $(\lambda x.R)S$ . The two sets of terms have type **tm** and **mtm** (standing for *term* and *marked term*) and are in Fig. 4.

There is no explicit constructor for variables. This point is a bit delicate. Let us try to explain it. The free variables are handled by the nabla quantifier (see the example in the next paragraph). The bound variables are provided by the HOAS-approach to binders, which codes binders as functions from terms to terms. For instance the term  $\lambda x.xx$  is represented as **abs**  $x \backslash$  **app**  $x\ x$ , *i.e.*

<code>kind tm</code>	<code>type.</code>	<code>kind mtm</code>	<code>type.</code>
<code>type app</code>	<code>tm -&gt; tm -&gt; tm.</code>	<code>type mapp</code>	<code>mtm -&gt; mtm -&gt; mtm.</code>
<code>type abs</code>	<code>(tm -&gt; tm) -&gt; tm.</code>	<code>type mabs</code>	<code>(mtm -&gt; mtm) -&gt; mtm.</code>
		<code>type mredex</code>	<code>(mtm -&gt; mtm) -&gt; mtm -&gt; mtm.</code>

**Fig. 4.** Definition of  $\lambda$ -terms and marked  $\lambda$ -terms in Abella

it is obtained by applying the `abs` constructor to the function  $x \mapsto \mathbf{app} \ x \ x$  (which maps the generic term  $x$  to the term `app x x`), coded in Abella with `x \ app x x`. In particular, `mabs M` and `mredex M N` are both of type `mtm`, but their subexpression `M` is not a term of type `mtm`, but rather a term of type `mtm->mtm`.

Finally, given a  $\beta$ -redex `app (abs M) N` representing  $(\lambda x.M)N$ , the reduct  $M\{x/N\}$  is denoted by `M N`, *i.e.* the application of the function `M` to `N` (application is given by juxtaposition). The user can forget any trouble with  $\alpha$ -equivalence and substitution: Abella takes care of them.

*An example.* Suppose that we want to write the predicate `tm M` which isolates terms without marked redexes in the larger set of marked terms. In Abella it can be written as follows:

```
Define tm : mtm -> prop by
  nabla x, tm x;
  tm (mabs M) := nabla x, tm (M x);
  tm (mapp M N) := tm M /\ tm N.
```

It corresponds exactly to the common way of defining  $\lambda$ -terms. The first line reads: *if  $x$  is a fresh variable then  $x$  is a term*. The second line: *`mabs M` is a term if given a fresh variable  $x$  the term obtained by applying the function `M` to  $x$  is a term*. Informally, one would simply ask that `M` is a term. What we used is nothing but its HOAS formulation, which has to adapt the informal approach because `M` itself is not of type `mtm` (but `mtm->mtm`). The third line: *`mapp M N` is a term if both `M` and `N` are terms*.

*Prism-diamond property.* Figure 5 contains the development of the prism-diamond property, where parallel reduction is the predicate `pred`.

Look at the second and the fourth cases of the definition of `pred`. There, `T` and `T'` are functions representing the binders associated to the corresponding abstractions. In the hypothesis of the two rules they are applied to `x`, in order to get a term. Moreover, the fourth case contains `T' U'`, the application of the function `T'` to `U'`: it is where  $\beta$ -reduction and substitution take place.

The substitution lemma follows, proved by simple induction on  $T \Rightarrow T'$ . Note that `T` is assumed to be a function, since in the conclusion we want to substitute `U` in `T` (and `U'` in `T'`). This is why the first assumption contains `(T x)`, and `x` is bound by  $\nabla$  (`nabla`). The commands `induction on 1`, `intros`, and `case H1` are the Abella code to start an induction on the first hypothesis. Then every line is

```

Define pred : tm -> tm -> prop by
  nabla x, pred x x;
  pred (abs T) (abs T') := nabla x, pred (T x) (T' x);
  pred (app T U) (app T' U') := pred T T' /\ pred U U';
  pred (app (abs T) U) (T' U') := nabla x, pred (T x) (T' x) /\ pred U U'.

Theorem pred_sub : forall T T' U U', nabla x,
  pred (T x) (T' x) -> pred U U' -> pred (T U) (T' U').
  induction on 1. intros. case H1.
  search.
  search.
  apply IH to H3 H2. search.
  apply IH to H3 H2. apply IH to H4 H2. search.
  apply IH to H3 H2. apply IH to H4 H2. search.

Theorem prism-diamond : forall T U1 U2,
  pred T U1 -> pred T U2 -> exists V, pred U1 V /\ pred U2 V /\ pred T V.
  induction on 1. intros. case H1.
  case H2. search.
  case H2. apply IH to H3 H4. search.
  case H2.
    apply IH to H3 H5. apply IH to H4 H6. search.
    case H3. apply IH to H4 H6. apply IH to H7 H5.
      apply pred_sub to H12 H9. search.
  case H2.
    case H5. apply IH to H3 H7. apply IH to H4 H6.
      apply pred_sub to H8 H11. search.
    apply IH to H3 H5. apply IH to H4 H6. apply pred_sub to H7 H10.
      apply pred_sub to H8 H11. search.

```

**Fig. 5.** Abella development of the prism-diamond property for  $t \Rightarrow t'$  (called `pred T T'`)

a case of the induction. The `search` tactic attempts to prove the current goal by an automatic simple search. Clearly, `apply IH` applies the inductive hypothesis.

The diamond property is proved by induction on the first hypothesis and case analysis of the second (note that each subcase starts with `case H2`). The proof uses only the *i.h.* and the substitution lemma, and it is the same proof which appears in [27], which also contains the only complete proof of confluence for  $\lambda$ -calculus based on HOAS of which we are aware.

*Prism-cube property.* The development for the prism-cube property is in the last page, after the bibliography, where  $R \xRightarrow{P} R'$  is represented with `res R P R'`. As we explained in Section 2 it follows exactly the same pattern used for the diamond property (plus the additional definition for the union of marked terms). The third component (`R' S'`) of the last two cases defining `res` is where  $\beta$ -reduction and substitution are used.

Beyond Huet's original paper [17], the cube property has also been formalized in [31,35]. Our development is the first one using HOAS, and it is sensibly simpler and shorter than the others. Indeed, it fits—in full—into one single page. We believe that this is quite remarkable.

## 4 Beyond the Pearl

This section contains a few observations and variations over our developments, others can be found in [1].

*Confluence by developments.* The (complete) development  $t^\circ$  of a term  $t$  is the result of the parallel step which reduces all the redexes in  $t$ . The development  $t^\circ$  can easily be described by induction on  $t$ , as follows:

$$\begin{array}{llll} x^\circ & = & x & (tu)^\circ & = & t^\circ u^\circ & \text{if } t \neq \lambda x.t' \\ (\lambda x.t)^\circ & = & \lambda x.t^\circ & ((\lambda x.t')u)^\circ & = & t'^\circ \{x/u^\circ\} \end{array}$$

Developments enjoy the following property, which is a sort of maximal prism property: if  $t \Rightarrow u$  then  $t \Rightarrow t^\circ$  and  $u \Rightarrow t^\circ$ . By specializing the prism-diamond property to complete developments, one gets the following *development property*:  $s_2 \leftarrow s \Rightarrow s_1$  implies  $s_1 \Rightarrow s^\circ \leftarrow s_2$  and  $s \Rightarrow s^\circ$ .

The proof of confluence by developments consists in showing the diamond property by proving the development property. This approach is dual to the use of residuals. Indeed, developments give a sort of *maximum* closure of any span  $u_2 \leftarrow t \rightarrow u_1$ , while residuals give the *minimum*.

The development property can be proved in Abella essentially as the prism-diamond property (see [1] for details). The formal proof (2 lemmas) is a sensible simplification of the one in Abella by Randy Pollack (18 lemmas), or of the similar one done in Twelf by Dan Licata. They can be found on the websites of the respective proof assistants, and both are based on Takahashi's proof [33] (see also [30]).

*Using the specification logic.* We now describe the impact of the specification logic provided by Abella on our developments. Let us come back to the toy  $\mathbf{tm}$  predicate of Section 3, which isolates terms among marked terms. At the specification level it takes the following form (the syntax of the specification level is different, an explanation follows):

```
 $\mathbf{tm} \text{ (mabs } R) \text{ :- pi } x \backslash \mathbf{tm} \ x \Rightarrow \mathbf{tm} \ (R \ x) .$ 
 $\mathbf{tm} \text{ (mapp } M \ N) \text{ :- } \mathbf{tm} \ M, \mathbf{tm} \ N .$ 
```

where  $\text{pi } x \backslash$  means  $\forall x$ , and  $\mathbf{tm} \ M, \mathbf{tm} \ N$  stays for  $\mathbf{tm} \ M$  and  $\mathbf{tm} \ N$ . There are two main differences. The first is that there is no case for free variables. This is due to the fact that  $\nabla$  is not in the weaker specification logic. It means that we can only represent bound variables, *i.e.* only closed terms. This fact induces the second difference: the first line of the definition says that for proving that  $\mathbf{abs} \ R$  is a term we need to prove that  $R \ x$  is a term *under the assumption that  $x$  is a term*. This new *under the assumption* part has a consequence at the reasoning level: it forces to annotate every use of a predicate involving binders with a *context*, *i.e.* the set of current assumptions under which the predicate holds. Such contexts are what allows to deal with open terms in this weaker setting without nabla. The idea is that to prove  $\mathbf{tm} \ (\mathbf{abs} \ R)$  we now need to be able to



prove the judgement  $\text{tm } x \vdash \text{tm } (R \ x)$  (where  $\vdash$  is the concrete notation for the turnstile  $\vdash$ ), *i.e.*  $\text{tm } (R \ x)$  in the context of assumptions  $\text{tm } x$ .

The presence of contexts usually requires to prove some properties about them, to introduce relations between contexts, and to generalize the statements of lemmas and theorems. In [1] we also recast our developments at the specification level. The structure of the proofs is the same, but some complications (*i.e.* additional definitions and lemmas) about contexts arise. Such complications are typically raised by statements having more than one predicate on the same term: the contexts of these predicates have to be related because they refer to the same binding structure. Indeed, the prism-diamond property (which uses only  $\Rightarrow$ ) requires less reasoning on contexts than the development and prism-cube properties (both using two predicates); this is also why in [27] worlds (the analogous of contexts in Twelf) do not require much attention, while they do require it in the proof of the development property by Dan Licata.

One of the aims of this paper is to show that in untyped frameworks the combined use of  $\nabla$  and HOAS gets formalizations which are extremely faithful to common pen-and-paper reasoning. Some of the examples dealing with the untyped  $\lambda$ -calculus on Abella website have been *lifted* from the specification to the reasoning logic, getting quite simpler and more readable formalizations, see [1]. It has to be said, however, that there are some untyped specifications that are of an intrinsic *closed nature*. For instance, the examples *equivalence of terms based on paths* and *determinism of translation between HOAS and de Bruijn representations* on the Abella website do not lift in a natural way to the reasoning level.

*Substitution lemmas.* In our developments we prove explicitly substitution lemmas for  $\Rightarrow$  and  $\xRightarrow{P}$ . One of the features of the specification logic is that it allows to get substitution lemmas for free. One would then expect that switching to the specification logic such lemmas disappear, and the formalizations get even shorter. Interestingly, this is not the case. Let us focus on  $\Rightarrow$ , which is simpler. The clause involving abstractions is:

$$\text{pred } (\text{abs } U) (\text{abs } V) \text{ :- } \text{pi } x \backslash \text{pred } x \ x \Rightarrow \text{pred } (U \ x) (V \ x).$$

This implies that the contexts for `pred` contains assumptions of the form `pred x x`. The substitution lemma provided by Abella then says that if  $t \Rightarrow t'$  then  $t\{x/u\} \Rightarrow t'\{x/u\}$ . This is not Lemma [1], because the second term should be  $t'\{x/u'\}$  (with  $u \Rightarrow u'$ ). But the assumption `pred x x` forces  $u' = u$ . Unfortunately, this simpler lemma cannot be used to prove the diamond property. It seems that it is enough to define `pred` as follows:

$$\text{pred } (\text{abs } U) (\text{abs } V) \text{ :- } \text{pi } x \backslash \text{pi } y \backslash \text{pred } x \ y \Rightarrow \text{pred } (U \ x) (V \ y).$$

One gets indeed the right substitution lemma. But now it is necessary to prove the reflexivity of `pred` (which before followed implicitly by the definition), which is non-trivial with assumptions of the form `pred x y`.

Summing up, the substitution lemmas for free provided by the specification level do not help in any way in the development under study.

**Acknowledgements.** To Dale Miller, who supervised me, encouraged me and helped me all along this work. To Kaustuv Chaudhuri and David Baelde for help and discussions about HOAS and Abella. To Fabien Renaud, Stéphane Zimmermann, and the anonymous reviewers for suggesting useful improvements. This work was partially supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168.

## References

1. Accattoli, B.: Sources, <https://sites.google.com/site/beniaminoaccattoli/residuals>
2. Baelde, D.: On the expressivity of minimal generic quantification. *Electr. Notes Theor. Comput. Sci.* 228, 3–19 (2009)
3. Barendregt, H.P.: *The Lambda Calculus – Its Syntax and Semantics*, vol. 103. North-Holland (1984)
4. Berry, G., Lévy, J.J.: Minimal and optimal computations of recursive programs. In: *POPL*, pp. 215–226 (1977)
5. Brotherston, J., Vestergaard, R.: A formalised first-order confluence proof for the  $\lambda$ -calculus using one-sorted variable names. *Inf. Comput.* 183(2), 212–244 (2003)
6. Dunfield, J., Pientka, B.: Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In: Giesl, J., Hähnle, R. (eds.) *IJCAR 2010*. LNCS, vol. 6173, pp. 15–21. Springer, Heidelberg (2010)
7. Elliott, C., Pfenning, F.: Higher-order abstract syntax. In: *PLDI*, pp. 199–208 (1988)
8. Gacek, A.: The Abella Interactive Theorem Prover (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) *IJCAR 2008*. LNCS (LNAI), vol. 5195, pp. 154–161. Springer, Heidelberg (2008)
9. Gacek, A.: A framework for specifying, prototyping, and reasoning about computational systems. Ph.D. thesis, University of Minnesota (September 2009)
10. Gacek, A.: Relating nominal and higher-order abstract syntax specifications. In: *PPDP 2010*, pp. 177–186. ACM (July 2010)
11. Gacek, A., Miller, D., Nadathur, G.: Combining generic judgments with recursive definitions. In: *LICS*, pp. 33–44 (2008)
12. Gacek, A., Miller, D., Nadathur, G.: Reasoning in Abella about structural operational semantics specifications. *ENTCS* 228, 85–100 (2009)
13. Gacek, A., Miller, D., Nadathur, G.: Nominal abstraction. *Inf. Comput.* 209(1), 48–73 (2011)
14. Gacek, A., Miller, D., Nadathur, G.: A two-level logic approach to reasoning about computations. *J. Autom. Reasoning* 49(2), 241–273 (2012)
15. Glauert, J.R.W., Khasidashvili, Z.: Relating conflict-free stable transition and event models via redex families. *Theor. Comput. Sci.* 286(1), 65–95 (2002)
16. Homeier, P.V.: A proof of the Church-Rosser theorem for the  $\lambda$ -calculus in higher order logic. In: *TPHOLs 2001: Supplemental Proceedings*, pp. 207–222 (2001)
17. Huet, G.P.: Residual theory in  $\lambda$ -calculus: A formal development. *J. Funct. Program.* 4(3), 371–394 (1994)

18. Huet, G.P., Lévy, J.J.: Computations in orthogonal rewriting systems, I. In: Computational Logic - Essays in Honor of Alan Robinson, pp. 395–414 (1991)
19. Huet, G.P., Lévy, J.J.: Computations in orthogonal rewriting systems, II. In: Computational Logic - Essays in Honor of Alan Robinson, pp. 415–443 (1991)
20. Lévy, J.J.: Réductions correctes et optimales dans le lambda-calcul. Thèse d'Etat, Univ. Paris VII, France (1978)
21. McKinna, J., Pollack, R.: Pure Type Systems Formalized. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 289–305. Springer, Heidelberg (1993)
22. Mellès, P.-A.: Axiomatic Rewriting Theory VI Residual Theory Revisited. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 24–50. Springer, Heidelberg (2002)
23. Miller, D., Nadathur, G.: A logic programming approach to manipulating formulas and programs. In: SLP, pp. 379–388 (1987)
24. Miller, D., Tiu, A.: A proof theory for generic judgments. *ACM Trans. Comput. Log.* 6(4), 749–783 (2005)
25. Miller, D., Tiu, A.: Proof search specifications of bisimulation and modal logics for the  $\pi$ -calculus. *ACM Trans. Comput. Log.* 11(2) (2010)
26. Nipkow, T.: More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 733–747 (1996)
27. Pfenning, F.: A proof of the Church-Rosser theorem and its representation in a logical framework. Tech. Rep. CMU-CS-92-186, Carnegie Mellon University (1992)
28. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
29. Pientka, B.: Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 1–12. Springer, Heidelberg (2010)
30. Pollack, R.: Polishing up the Tait-Martin-Löf proof of the Church-Rosser theorem (1995)
31. Rasmussen, O.: The Church-Rosser theorem in Isabelle: a proof porting experiment. Tech. Rep. 164, University of Cambridge (1995)
32. Shankar, N.: A mechanical proof of the Church-Rosser theorem. *J. ACM* 35(3), 475–522 (1988)
33. Takahashi, M.: Parallel reductions in  $\lambda$ -calculus. *Inf. Comput.* 118(1), 120–127 (1995)
34. Terese: Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
35. Vestergaard, R.: The Primitive Proof Theory of the lambda-Calculus. Ph.D. thesis, Heriot-Watt University, Edinburgh, Scotland (2003)

```

Define res : mtm -> mtm -> mtm -> prop by
  nabla x, res x x x;
  res (mabs R) (mabs P) (mabs R') := nabla x, res (R x) (P x) (R' x);
  res (mapp R S) (mapp P Q) (mapp R' S') := res R P R' /\ res S Q S';
  res (mredex R S) (mapp (mabs P) Q) (mredex R' S') :=
    nabla x, res (R x) (P x) (R' x) /\ res S Q S';
  res (mapp (mabs R) S) (mredex P Q) (R' S') :=
    nabla x, res (R x) (P x) (R' x) /\ res S Q S';
  res (mredex R S) (mredex P Q) (R' S') :=
    nabla x, res (R x) (P x) (R' x) /\ res S Q S'.

Define res_union : mtm -> mtm -> mtm -> prop by
  nabla x, res_union x x x;
  res_union (mabs R) (mabs P) (mabs Q) := nabla x, res_union (R x) (P x) (Q x);
  res_union (mapp R S) (mapp P T) (mapp Q U) :=
    res_union R P Q /\ res_union S T U;
  res_union (mredex R S) (mredex P T) (mredex Q U) :=
    nabla x, res_union (R x) (P x) (Q x) /\ res_union S T U;
  res_union (mapp (mabs R) S) (mredex P T) (mredex Q U) :=
    nabla x, res_union (R x) (P x) (Q x) /\ res_union S T U;
  res_union (mredex R S) (mapp (mabs P) T) (mredex Q U) :=
    nabla x, res_union (R x) (P x) (Q x) /\ res_union S T U.

Theorem res_subst : forall R P R' S Q S', nabla x,
  res (R x) (P x) (R' x) -> res S Q S' -> res (R S) (P Q) (R' S').
  induction on 1. intros. case H1.
  search.
  search.
  apply IH to H3 H2. search.
  apply IH to H3 H2. apply IH to H4 H2. search.
  apply IH to H3 H2. apply IH to H4 H2. search.
  apply IH to H3 H2. apply IH to H4 H2. search.
  apply IH to H3 H2. apply IH to H4 H2. search.

Theorem prism_cube : forall Q P R Q' Q'',
  res Q R Q'' -> res Q P Q' -> exists P' R' RunionP S,
  res P R P' /\ res R P R' /\ res Q' R' S /\
  res Q'' P' S /\ res Q RunionP S /\ res_union P R RunionP.
  induction on 1. intros. case H1.
  case H2. search.
  case H2. apply IH to H3 H4. search.
  case H2.
    apply IH to H3 H5. apply IH to H4 H6. search.
    case H3. apply IH to H4 H6. apply IH to H7 H5.
      apply res_subst to H16 H10. search.
  case H2.
    apply IH to H3 H5. apply IH to H4 H6. search.
    apply IH to H3 H5. apply IH to H4 H6. apply res_subst to H9 H15. search.
  case H2.
    case H5. apply IH to H3 H7. apply IH to H4 H6.
      apply res_subst to H11 H17. search.
    apply IH to H3 H5. apply IH to H4 H6. apply res_subst to H9 H15.
      apply res_subst to H10 H16. search.
  case H2.
    apply IH to H3 H5. apply IH to H4 H6. apply res_subst to H10 H16. search.
    apply IH to H3 H5. apply IH to H4 H6. apply res_subst to H9 H15.
      apply res_subst to H10 H16. search.

```

# A String of Pearls: Proofs of Fermat’s Little Theorem

Hing-Lun Chan<sup>1</sup> and Michael Norrish<sup>2</sup>

<sup>1</sup> Australian National University  
joseph.chan@anu.edu.au

<sup>2</sup> Canberra Research Lab, NICTA\*;  
also, Australian National University  
Michael.Norrish@nicta.com.au

**Abstract.** We discuss mechanised proofs of Fermat’s Little Theorem in a variety of styles, focusing in particular on an elegant combinatorial “necklace” proof that has not been mechanised previously. What is elegant in prose turns out to be long-winded mechanically, and so we examine the effect of explicitly appealing to group theory. This has pleasant consequences both for the necklace proof, and also for the direct number-theoretic approach.

## 1 Introduction

Fermat’s Little Theorem is a famous result in basic number theory. When  $p$  is prime, then

$$a^p \equiv a \pmod{p} \quad \text{for any natural number } a.$$

Though resources like Wikipedia [15] provide an extensive range of proofs of this result, it seems that standard practice in interactive proof assistants (e.g. Hurd *et al.* [9]) is to use Euler’s generalisation, which is number-theoretic. There is good reason for this: the number theory required is actually quite simple, making it easy to establish the result without needing a great deal of background theory. This paper shows, however, how a number of other proofs, some with interesting ideas, can be performed mechanically.

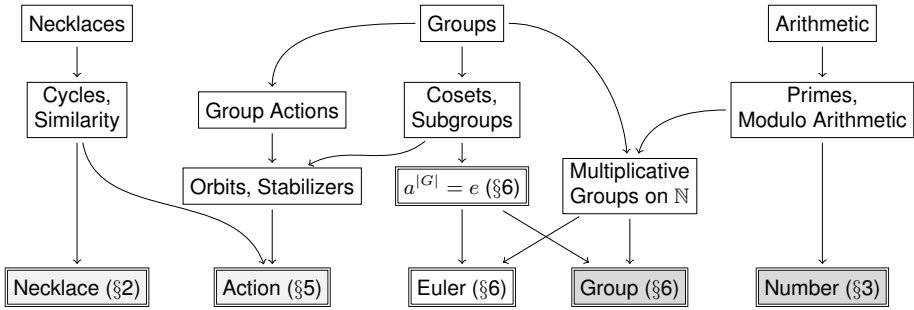
The simplest such proof (the necklace) is based on combinatorics over lists. It is relatively straightforward to mechanise (we use the HOL4 proof assistant [13], which has built-in support for lists), but aspects of the proof become smoother when it is rephrased in the language of group theory. This required background does not represent a particularly onerous burden for mechanisation. Indeed, the more group theory one has to hand, the more polished the proof becomes.

We also examine the effect of using group theory in number-theoretic approaches.

**Overview.** The rest of the paper is structured as follows. In Sections 2 and 3, we describe both the standard number-theoretic proof, and Golomb’s combinatorial necklace proof [4], and their mechanisation. In Section 4, we discuss how the required (rather basic) group theory is mechanised, before showing how this theory can be applied to

---

\* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.



**Fig. 1.** Theory dependencies for proofs of Fermat’s Little Theorem. Double-lined boxes indicate significant results discussed in the corresponding section of the paper. The leftmost **Necklace** and rightmost **Number** are direct proofs; others use group theory. Combinatoric results (in light gray) are of  $a^p \equiv a \pmod p$ , which is equivalent (when  $0 < a < p$ ) to number-theoretic results (in dark gray) of  $a^{p-1} \equiv 1 \pmod p$ . **Euler** is  $a^{\varphi(n)} \equiv 1 \pmod n$ .

the necklace proof (in Section 5), and to the number-theoretic proof (in Section 6). We conclude in Section 7 including a comparison of the different approaches in terms of their complexity.

Figure 1 gives a graphical summary of the logical dependencies underlying all of the proofs we discuss. The graph falls into three parts:

- The leftmost route shows Golomb’s necklace proof, a combinatorial proof based on rotations of lists.
- The rightmost route shows an elementary number-theoretic proof commonly found in textbooks.
- The central paths are of various group-theoretic proofs. The first path from the left is the application of group theory (via the notions of action, orbit, and stabilizer) to the necklace proof. The others show the proof of the group-theoretic analogue of Fermat’s Little Theorem, followed by the derivation of specific results in the domain of natural numbers.

**HOL4 Notation and Theorems.** All statements appearing with a turnstile ( $\vdash$ ) are HOL4 theorems, automatically pretty-printed to  $\LaTeX$  from the relevant theory in the HOL4 development. Notation specific to this paper is explained as it is introduced. Otherwise, HOL4 supports a notation that is a generally pleasant combination of quantifiers ( $\forall, \exists$ ) and functional programming ( $\lambda$  for function abstraction, juxtaposition for function application).

Lists are written between square brackets, e.g.,  $[1; 2]$ . The length of a list  $\ell$  is written  $|\ell|$ . The concatenation of  $\ell_1$  and  $\ell_2$  is written  $\ell_1 ++ \ell_2$ . Sets are written between braces, also allowing comprehensions such as  $\{x \mid x < 6\}$ . Sets also support standard operations such as cardinality (also written with vertical bars:  $|\{3; 5\}| = 2$ ), union ( $\cup$ ), intersection ( $\cap$ ), and difference ( $\setminus$ ). We write  $\text{IMAGE } f \ s$  for the image of the set  $s$  under function  $f$ , and  $\text{BIJ } f \ s_1 \ s_2$  means that function  $f$  is a bijection

between sets  $s_1$  and  $s_2$ . The term  $R$  `equiv_on`  $s$  means that  $R$  is an equivalence relation on the set  $s$ , and `partition`  $R$   $s$  denotes the set of subsets of  $s$  that are partitions with respect to an equivalence relation  $R$ .

**Our Contribution.** As already noted, Fermat’s Little Theorem has been mechanised a number of times before, *e.g.*, in Coq [10], ACL2 [12] and HOL Light [6]. The minimal group theory we used and mechanised is also very standard.<sup>1</sup> Our contribution is the mechanisation of the necklace proof, in direct and group-theoretic styles (we believe both to be entirely novel). We also compare these proofs with the standard number-theoretic approaches.

**Availability.** HOL4 proof scripts can be found at <http://bitbucket.org/jhlchan/hol/src>. The linearised scripts (discussed in Section 7) are those beginning with prefix `All` in the `fermat` directory. Proofs as they were developed (in various separate theories) are laid out in sub-directories below `fermat`.

## 2 The Necklace Proof

Leonard Eugene Dickson, in his authoritative treatise *History of the Theory of Numbers* [3, Chapter 3], thoroughly documented all known proofs of this Fermat’s result, up to 1919. Among them was this nice combinatorial proof by Julius Petersen in 1872:

*Take  $p$  elements from  $q$  with repetitions in all ways, that is, in  $q^p$  ways. The  $q$  sets with elements all alike are not changed by a cyclic permutation of the elements, while the remaining  $q^p - q$  sets are permuted in sets of  $p$  [when  $p$  is prime]. Hence  $p$  divides  $q^p - q$ .*

This idea is the basis of the *Necklace proof* of S. W. Golomb [4], which has since been re-discovered or discussed by many others (*e.g.*, Smyth [14], Rouse [11], Evans [7], and Conrad [2]).

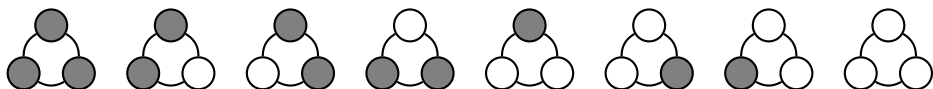
The necklaces of Golomb’s proof are the  $p$  elements drawn from a set of cardinality  $q$ . Where Peterson has cyclic permutations, Golomb’s version adds the image of rotating beads on a necklace (Figure 2).

### 2.1 Necklaces and Colours

Consider the set  $N$  of necklaces of length  $n$  (*i.e.*,  $n$  beads) with  $a$  colours (*i.e.*,  $a$  choices for a bead’s colour). Since a bead can have any of the  $a$  colours, and there are  $n$  beads in total, the total number of necklaces is  $|N| = a^n$ , or  $|N| = a^n$ . Of these necklaces, the *monocoloured* necklaces are those with the same colour for all beads; the others are *multicoloured* necklaces.

Let  $S$  (for single) denote the set of monocoloured necklaces, and  $M$  (for multiple) denote the multi-coloured necklaces. Clearly,  $N = S \cup M$ , and  $S \cap M = \emptyset$  — that is,

<sup>1</sup> The Orbit-Stabiliser theorem has not been mechanised before in HOL, but this is a minor contribution given the existing work in other systems such as Coq.



**Fig. 2.** Necklaces with 3 beads in 2 colours. The first and last are *monocoloured* necklaces. The other *multicoloured* necklaces are divided into 2 parts: those with one white bead and those with two white beads. Multicoloured necklaces in each part can cycle only among themselves. Note that, for 3 beads, each part consists of 3 necklaces. Hence the number of multi-2-coloured necklaces with 3 beads (which is  $2^3 - 2 = 8 - 2 = 6$ ) is divisible by 3.

$S$  and  $M$  are disjoint, and they form a partition of the set of necklaces  $N$ . Since there is only 1 monocoloured necklace for each colour, the number of monocoloured necklaces  $|S|$  is just  $a$ . Given that the two types of necklaces partition the whole set, the number of multicoloured necklaces  $|M|$  is equal to  $|N| - |S| = a^n - a$ .

*HOL Implementation.* Let  $(\text{necklace } n \ a)$  be the set of necklaces of length  $n$  with  $a$  colours. The HOL definition is

$$\vdash \text{necklace } n \ a = \{ \ell \mid |\ell| = n \wedge \text{set } \ell \subseteq \text{count } a \}$$

Our necklaces' beads are just natural numbers, and the definition requires that the "colours" of the necklace are simply drawn from the set  $(\text{count } a)$ : the set of natural numbers less than  $a$ .

Simple properties of the set  $(\text{necklace } n \ a)$  readily follow from the definition:

$$\begin{aligned} \vdash & \text{FINITE } (\text{necklace } n \ a) \\ \vdash & |\text{necklace } n \ a| = a^n \end{aligned}$$

The monocoloured and multicoloured necklaces are defined thus:

$$\begin{aligned} \vdash & \text{monocoloured } n \ a = \{ \ell \mid \ell \in \text{necklace } n \ a \wedge (\ell \neq [] \Rightarrow \text{SING } (\text{set } \ell)) \} \\ \vdash & \text{multicoloured } n \ a = \text{necklace } n \ a \setminus \text{monocoloured } n \ a \end{aligned}$$

where  $\text{SING } (\text{set } \ell)$  means that the set of list elements  $(\text{set } \ell)$  is a singleton. The cardinality results for these sets are straightforward:

$$\begin{aligned} \vdash & 0 < n \Rightarrow |\text{monocoloured } n \ a| = a \\ \vdash & 0 < n \Rightarrow |\text{multicoloured } n \ a| = a^n - a \end{aligned}$$

In order to show that the last expression  $a^n - a$  is divisible by  $n$  when length  $n$  is prime, we need to know something more about the multicoloured necklaces, especially how an equivalence relation involving cyclic permutations partitions the set.

## 2.2 Cycles

Necklaces are represented by lists of length  $n$ . Following the imagery, it is natural to think of them as being joined from end to end. We define a `cycle` operation on lists:



$$\vdash \text{cycle } n \ell = \text{FUNPOW } (\lambda \ell. \text{DROP } 1 \ell ++ \text{TAKE } 1 \ell) \ n \ \ell$$

The expression  $\text{DROP } n \ell$  discards the first  $n$  elements of list  $\ell$ , returning whatever remains, while  $\text{TAKE } n \ell$  returns the first  $n$  elements (for the empty list  $[\ ]$ ,  $\text{TAKE}$  and  $\text{DROP}$  both return  $[\ ]$ ). By putting  $n = 1$ , this is chopping off the first bead, shifting it to the other end and adding it back. Therefore  $\text{DROP } 1 \ell ++ \text{TAKE } 1 \ell$  represents a rotation by 1 bead position. Then  $\text{FUNPOW}$  just repeats this operation  $n$  times.<sup>2</sup> These elementary facts about  $\text{cycle}$  follow immediately:

$$\begin{aligned} \vdash \text{cycle } 0 \ell &= \ell && (\text{CYCLE\_0}) \\ \vdash \text{cycle } n (\text{cycle } m \ell) &= \text{cycle } (n + m) \ell && (\text{CYCLE\_ADD}) \end{aligned}$$

Applying  $\text{cycle}$  on a necklace results in another necklace, of the same length and colours:

$$\begin{aligned} \vdash \ell \in \text{necklace } n \ a &\Rightarrow \forall k. \text{cycle } k \ell \in \text{necklace } n \ a \\ \vdash |\text{cycle } n \ell| &= |\ell| \\ \vdash \text{set } (\text{cycle } n \ell) &= \text{set } \ell \end{aligned}$$

As a result,  $\text{cycle}$  of a monocoloured necklace is still monocoloured, and  $\text{cycle}$  of a multicoloured necklace is still multicoloured, as expected.

We can reason about cycles with modular arithmetic:

$$\begin{aligned} \vdash \ell \neq [\ ] &\Rightarrow \text{cycle } n \ell = \text{cycle } (n \bmod |\ell|) \ell && (\text{CYCLE\_MOD\_LENGTH}) \\ \vdash \ell \neq [\ ] &\Rightarrow \text{cycle } m (\text{cycle } n \ell) = \text{cycle } ((m + n) \bmod |\ell|) \ell \end{aligned}$$

And ultimately, a cycle can come full-circle, in multiples, or can be undone by another cycle:

$$\begin{aligned} \vdash \text{cycle } |\ell| \ell &= \ell && (\text{CYCLE\_BACK}) \\ \vdash \text{cycle } n \ell = \ell &\Rightarrow \forall m. \text{cycle } (m \times n) \ell = \ell && (\text{CYCLE\_MULTIPLE\_BACK}) \\ \vdash n \leq |\ell| &\Rightarrow \text{cycle } (|\ell| - n) (\text{cycle } n \ell) = \ell && (\text{CYCLE\_INV}) \end{aligned}$$

Already, one can see the possible connections to group theory.

### 2.3 Similarity and Partitions

We shall say two necklaces  $\ell_1, \ell_2$  are *similar*, denoted  $\ell_1 == \ell_2$ , when:

$$\vdash \ell_1 == \ell_2 \iff \exists n. \ell_2 = \text{cycle } n \ell_1$$

That is,  $\ell_1$  can cycle to  $\ell_2$  because they consist of the same beads in cyclic order.

The following properties of  $(==)$  follow from properties of  $\text{cycle}$ :

$$\begin{aligned} \vdash \ell == [\ ] \vee [\ ] == \ell &\iff \ell = [\ ] \\ \vdash \ell_1 == \ell_2 &\Rightarrow |\ell_1| = |\ell_2| \end{aligned}$$

<sup>2</sup> This definition of  $\text{cycle } n \ell$  using  $\text{FUNPOW}$  makes sense for all  $n$ , whereas a definition using  $\text{TAKE } n$  and  $\text{DROP } n$  would only work when  $n \leq |\ell|$ .

With a little more effort, the fact that  $(=)$  is an equivalence relation can be proved:

$$\begin{aligned} &\vdash \ell == \ell \\ &\vdash \ell_1 == \ell_2 \Rightarrow \ell_2 == \ell_1 \\ &\vdash \ell_1 == \ell_2 \wedge \ell_2 == \ell_3 \Rightarrow \ell_1 == \ell_3 \end{aligned}$$

The key for reflexivity is `CYCLE_0`, for symmetry is `CYCLE_INV`, for transitivity is `CYCLE_ADD`.

Let us denote the equivalence classes under  $(=)$  by associates:

$$\vdash \text{associates } x = \{y \mid x == y\}$$

As  $(=)$  is an equivalence relation, the `associates` partition the set of necklaces. This partitioning has a particularly simple structure when the necklace length  $n$  is prime.

## 2.4 Multicoloured Necklaces with Prime Length

First, an important result about values that “cycle back” and their greatest common divisor:

**Theorem 1.** *If two values  $m, n$  can cycle back, the value  $\text{gcd } m \ n$  can also cycle back.*

$$\vdash \text{cycle } m \ \ell = \ell \wedge \text{cycle } n \ \ell = \ell \Rightarrow \text{cycle } (\text{gcd } m \ n) \ \ell = \ell$$

*Proof.* If  $n = 0$ , then  $\text{cycle } (\text{gcd } m \ 0) \ \ell = \text{cycle } m \ \ell = \ell$  by assumption. Otherwise, we can use Bézout’s identity, called `LINEAR_GCD` in HOL library, which states that if  $n \neq 0$ , then there exist  $p$  and  $q$  such that  $p \times n = q \times m + \text{gcd } m \ n$ , and reason:

$$\begin{aligned} &\text{cycle } (\text{gcd } m \ n) \ \ell \\ &= \text{cycle } (\text{gcd } m \ n) \ (\text{cycle } (q \times m) \ \ell) && \text{by CYCLE\_MULTIPLE\_BACK} \\ &= \text{cycle } (\text{gcd } m \ n + q \times m) \ \ell && \text{by CYCLE\_ADD} \\ &= \text{cycle } (p \times n) \ \ell && \text{by LINEAR\_GCD} \\ &= \ell && \text{by CYCLE\_MULTIPLE\_BACK} \end{aligned}$$

□

A distinguishing feature of monocoloured necklaces is:

**Theorem 2.** *A necklace  $\ell$  is monocoloured iff  $\text{cycle } 1 \ \ell = \ell$ .*

$$\begin{aligned} &\vdash 0 < n \wedge 0 < a \wedge \ell \in \text{necklace } n \ a \Rightarrow \\ &\quad (\ell \in \text{monocoloured } n \ a \iff \text{cycle } 1 \ \ell = \ell) \end{aligned}$$

*Proof.* A monocoloured necklace  $\ell$  has all beads the same colour, so shifting 1 bead makes no difference, hence  $\text{cycle } 1 \ \ell = \ell$ . Conversely, given  $\text{cycle } 1 \ \ell = \ell$ , applying `CYCLE_MULTIPLE_BACK`,  $\ell = \text{cycle } 2 \ \ell = \text{cycle } 3 \ \ell = \dots$ . As lists, head of  $\ell$  is the first bead, head of  $\text{cycle } 1 \ \ell$  is the second bead, head of  $\text{cycle } 2 \ \ell$  is the third bead, etc. Since these cycle lists are all the same, and equal lists mean equal heads, all beads have the same colour, making the necklace  $\ell$  monocoloured. □

We proceed to find the size of associates of multicoloured necklaces with prime length:

**Theorem 3.** *For multicoloured necklaces  $\ell$  with prime  $|\ell| = p$ , the cycle map from count  $p$  to associates  $\ell$  is injective.*

$$\vdash \text{prime } p \wedge \ell \in \text{multicoloured } p \ a \Rightarrow \\ \text{INJ } (\lambda n. \text{cycle } n \ \ell) \ (\text{count } p) \ (\text{associates } \ell)$$

*Proof.* This is to show that, for all  $x < p$  and  $y < p$ ,  $\text{cycle } x \ \ell = \text{cycle } y \ \ell \Rightarrow x = y$ . Suppose this is not the case. Then there are  $x \neq y$  such that there is a common  $\ell' = \text{cycle } x \ \ell = \text{cycle } y \ \ell$ . Note that both necklaces  $\ell'$  and  $\ell$  are multicoloured with same length  $p$  (Section 2.2). Without loss of generality, assume  $x < y$ . Then  $y = d + x$ , where difference  $d > 0$  and  $d < p$  (since both  $x < p$  and  $y < p$ ). Hence  $\text{cycle } d \ \ell' = \text{cycle } d \ (\text{cycle } x \ \ell) = \text{cycle } (d + x) \ \ell = \text{cycle } y \ \ell = \ell'$ . With  $\text{cycle } d \ \ell' = \ell'$ , and from CYCLE\_BACK we have  $\text{cycle } p \ \ell' = \ell'$ , hence  $\text{cycle } (\text{gcd } d \ p) \ \ell' = \ell'$  by Theorem 1. But  $\text{gcd } d \ p = 1$  for prime  $p, 0 < d < p$ . This implies the multicoloured necklace  $\ell'$  has  $\text{cycle } 1 \ \ell' = \ell'$ , which is a contradiction in view of Theorem 2.  $\square$

**Theorem 4.** *For multicoloured necklaces  $\ell$  with  $|\ell| = n$  (prime or non-prime), the cycle map from count  $n$  to associates  $\ell$  is surjective.*

$$\vdash \ell \in \text{multicoloured } n \ a \Rightarrow \text{SURJ } (\lambda k. \text{cycle } k \ \ell) \ (\text{count } n) \ (\text{associates } \ell)$$

*Proof.* This is because, if a necklace  $\ell'$  is similar to  $\ell$ , there is a  $k$  such that  $\ell' = \text{cycle } k \ \ell$ . By CYCLE\_MOD\_LENGTH,  $\ell' = \text{cycle } (k \bmod n) \ \ell$ , and so it is in the range of count  $n$ .  $\square$

**Theorem 5.** *For multicoloured necklaces  $\ell$  with prime  $|\ell| = p$ , their associates have size  $p$ .*

$$\vdash \text{prime } p \wedge \ell \in \text{multicoloured } p \ a \Rightarrow |\text{associates } \ell| = p$$

*Proof.* Since the cycle map is surjective in general (Theorem 4), and injective when the necklace length is prime (Theorem 3), there is a bijection between count  $p$  and associates  $\ell$  for multicoloured necklaces  $\ell$  when  $|\ell| = p$  is prime. The result follows from this bijection between finite sets.  $\square$

This leads directly to the following mechanisation of the necklace proof of

**Theorem 6.** *Fermat's Little Theorem.*

$$\vdash \text{prime } p \Rightarrow p \text{ divides } a^p - a$$

*Proof.* For prime  $p$ , the multicoloured necklaces  $\ell \in \text{multicoloured } p \ a$  are “permuted in sets of  $p$ ”, as claimed by Julius Petersen (Section 2), since  $|\text{associates } \ell| = p$  by Theorem 5. Recall that associates  $\ell$  are the equivalence classes of  $(=)$  on multicoloured  $p \ a$  (Section 2.3). Since equivalence classes form a partition, and here they all have the same size  $p$ , we have:

$$|\text{multicoloured } p \ a| = p \times |\text{partition } (=) \ (\text{multicoloured } p \ a)|$$

As  $p$  is prime,  $0 < p$ , and  $|\text{multicoloured } p \ a| = a^p - a$  (Section 2.1). Combining these results we have  $p$  divides  $a^p - a$  by definition of divides.  $\square$

### 3 Direct Number-Theoretic Proof

The proof of Fermat's Little Theorem given in most textbooks, and also that given in various theorem-proving systems, is number-theoretic, based on properties of modulo arithmetic. In particular, modulo prime multiplication has some special properties. The first one, usually referred to as Euclid's Lemma, is that a prime divides a product iff the prime divides a factor. In terms of modulo arithmetic, this is:

$$\vdash \text{prime } p \Rightarrow (x \times y \equiv 0 \pmod{p}) \iff x \equiv 0 \pmod{p} \vee y \equiv 0 \pmod{p})$$

Thus, left-cancellation of a non-zero factor is possible in prime modulo arithmetic:

$$\vdash \text{prime } p \wedge x \times y \equiv x \times z \pmod{p} \wedge x \not\equiv 0 \pmod{p} \Rightarrow y \equiv z \pmod{p}$$

**Definition 1.** Let the residues of prime  $p$  be the non-zero numbers less than  $p$ :  $\{1 \dots p-1\}$ .

Take any  $a$  from the residues of  $p$ , and consider the various values of  $a \times x \pmod{p}$ , for all  $x$  also a residue of  $p$ . In HOL, this is denoted by a row operation:

**Definition 2.**  $\vdash \text{row } p \ a \ x = a \times x \pmod{p}$

**Theorem 7.** The row products form a permutation of the residues for prime modulo.

$$\vdash \text{prime } p \wedge a \in \{1 \dots p-1\} \Rightarrow \{1 \dots p-1\} = \text{IMAGE } (\text{row } p \ a) \ \{1 \dots p-1\}$$

*Proof.* The IMAGE on the right-hand side is equivalent to  $\{a \times x \pmod{p} \mid 1 \leq x \wedge x < p\}$ . The possible remainders under modulo  $p$  are  $0, 1, \dots, p-1$ . Since a prime  $p$  has no proper factors, and both  $a$  and  $x$  are less than  $p$ , the product  $a \times x$  cannot be the prime  $p$ , nor any multiple of the prime  $p$ . Hence the remainder,  $a \times x \pmod{p}$  cannot be zero, making this result one of the residues of  $p$ . The possible values are distinct because if  $a \times x \equiv a \times y \pmod{p}$ , then  $x \equiv y \pmod{p}$  by left-cancellation of non-zero  $a$ . So the right-hand side, the row products, is just a permutation of the left-hand side, the residues of  $p$ .  $\square$

This is the key for the number-theoretic proof of

**Theorem 8.** Fermat's Little Theorem (equivalent form<sup>3</sup>)

$$\vdash \text{prime } p \wedge a \in \{1 \dots p-1\} \Rightarrow a^{p-1} \equiv 1 \pmod{p}$$

*Proof.* Consider multiplying all numbers (denoted by the symbol  $\prod$ ) from each of these finite sets:

(1) the residues  $\{1 \dots p-1\}$  and (2) its row products  $\text{IMAGE } (\text{row } p \ a) \ \{1 \dots p-1\}$ .

Clearly, the first one is a factorial:

---

<sup>3</sup> To show  $a^p \equiv a \pmod{p}$  for all  $a$ , it is sufficient to show this for  $0 \leq a < p$ , the possible remainders under modulo  $p$ . The case  $a = 0$  is trivial. The case  $0 < a < p$  has  $\text{gcd } a \ p = 1$ , since  $p$  is prime. This allows left-cancellation of non-zero  $a$  on both sides, giving the equivalent form  $a^{p-1} \equiv 1 \pmod{p}$ .

$$\vdash \prod \{1..p-1\} = (p-1)!$$

For the second one, since for numbers the order of multiplication does not affect the product, all the factors  $a$  of  $(\text{row } p \ a)$  (Definition 2) can be collected together, so we have:

$$\vdash \text{prime } p \wedge a \in \{1..p-1\} \Rightarrow \prod (\text{IMAGE } (\text{row } p \ a) \ \{1..p-1\}) \equiv a^{p-1} \times (p-1)! \pmod{p}$$

As the underlying sets are the same due to permutation (Theorem 7), the two products under modulo  $p$  are identical:

$$\vdash \text{prime } p \wedge a \in \{1..p-1\} \Rightarrow (p-1)! \equiv a^{p-1} \times (p-1)! \pmod{p}$$

A prime  $p$  has no proper factor, and  $(p-1)!$  has of all the numbers less than  $p$ , so  $(p-1)! \not\equiv 0 \pmod{p}$ . Applying non-zero left-cancellation of modulo  $p$  multiplication gives Fermat's Little Theorem.  $\square$

## 4 Group Theory

The combinatorial necklace proof and the number-theoretic proof may appear unrelated, but there is an underlying algebra behind both proofs, that of group theory. The algebra gives us:

- the cycles and similarities in the necklace proof;
- the factor cancellation in the number-theoretic proof; and
- an insight, allowing a modest generalisation.

The discussion that follows is an expansion of this theme.

We mechanise the necessary theorems from group theory following an existing mechanisation in the HOL distribution by Joe Hurd [9]. We have a predicate `Group`  $g$  on a record of four fields (the group operation  $(x \times y)$ , the inverse  $(x^{-1})$ , the identity  $(e)$  and the carrier set). We abuse notation to allow  $G$  and  $H$  to stand for the carrier sets of groups  $g$  and  $h$  respectively:

$$\begin{aligned} \vdash \text{Group } g \iff & \\ & e \in G \wedge (\forall x \ y :: (G). \ x \times y \in G) \wedge (\forall x :: (G). \ x^{-1} \in G) \wedge \\ & (\forall x :: (G). \ e \times x = x) \wedge (\forall x :: (G). \ x^{-1} \times x = e) \wedge \\ & \forall x \ y \ z :: (G). \ x \times y \times z = x \times (y \times z) \end{aligned}$$

The double-colon notation (e.g.,  $\forall x \ y \ z :: (G). \ P \ x \ y$ ) is a restriction on all the preceding bound variables ( $x$  and  $y$  here) requiring them to be in the set  $G$ .

Typical results in this mechanisation appear with the `Group` predicate as a side-condition.

$$\begin{aligned} \vdash \text{Group } g \Rightarrow \forall x \ y \ z :: (G). \ x \times y = x \times z \iff y = z \\ \vdash \text{Group } g \Rightarrow \forall x \ y \ z :: (G). \ x \times y = z \iff x = z \times y^{-1} \\ \vdash \text{Group } g \Rightarrow \forall x :: (G). \ (x^{-1})^{-1} = x \end{aligned}$$

<sup>4</sup> The source code of a prior HOL mechanisation of group theory by Elsa L. Gunter [5] is not generally available.

This is perhaps not the slickest possible presentation of abstract algebra, even within the constraints of HOL4's simple type theory, but it is both well-understood and sufficient for our purposes.

Group *exponentiation* is defined via primitive recursion, giving us the usual properties:

$$\begin{aligned} \vdash \text{Group } g \wedge x \in G &\Rightarrow x^0 = e \\ \vdash \text{Group } g \wedge x \in G &\Rightarrow x^1 = x \\ \vdash \text{Group } g \wedge x \in G &\Rightarrow x^{m \times n} = (x^m)^n \\ \vdash \text{Group } g \wedge x \in G &\Rightarrow (x^n)^{-1} = (x^{-1})^n \end{aligned}$$

We write  $h \leq g$  to mean that  $h$  is a subgroup of  $g$ , and define the *coset* of a set  $X$  with respect to a group element  $a$  (normally written  $aX$ ) to be

$$\vdash \text{coset } g \ X \ a = \text{IMAGE } (\lambda z. a \times z) \ X$$

The cosets of a subgroup's carrier are important because of these standard results:

**Theorem 9.** *Subgroup cosets partition the group's carrier set, by the following equivalence relation:*

$$\vdash \text{Group } g \wedge h \leq g \Rightarrow \text{coset } g \ H \ \text{equiv\_on } G$$

**Theorem 10.** *Each coset of a subgroup is in bijection with the subgroup itself.*

$$\vdash \text{Group } g \wedge h \leq g \wedge a \in G \Rightarrow \text{BIJ } (\lambda x. a \times x) \ H \ (\text{coset } g \ H \ a)$$

This bijection allows determination of the size of subgroup cosets:

**Theorem 11.** *For a finite subgroup, the size of its coset equals the size of subgroup itself:*

$$\vdash \text{Group } g \wedge h \leq g \wedge a \in G \wedge \text{FINITE } H \Rightarrow |\text{coset } g \ H \ a| = |H|$$

Therefore the subgroup cosets partition consists of equal-size chunks, leading to Lagrange's Identity:

$$\vdash \text{FiniteGroup } g \wedge h \leq g \Rightarrow |G| = |H| \times |\text{partition } (\text{coset } g \ H) \ G|$$

and Lagrange's Theorem on cardinality of subgroups:

$$\vdash \text{FiniteGroup } g \wedge h \leq g \Rightarrow |H| \ \text{divides} \ |G|$$

## 5 Group Theory Applied to the Necklace Proof

The group-theoretic version of the necklace proof requires a little more theory than the basic development of the preceding section. We shall use the group  $\mathbb{Z}_n^+$ , which is the additive group over the natural numbers less than  $n$ . This group's binary operation is addition modulo  $n$ , and its identity is zero.

## 5.1 Group Actions

**Definition 3.** Let  $g$  be a group over a set of elements of type  $\alpha$ ,  $X$  be a set of elements of type  $\beta$ , and (infix)  $\circ$  a function of type  $\alpha \rightarrow \beta \rightarrow \beta$ . The mapping  $(\circ)$  is called a group action from  $g$  to  $X$ , (written action  $(\circ) g X$ ), if these three conditions are satisfied:

- Closure:  $a \in G \wedge x \in X \Rightarrow a \circ x \in X$
- Identity:  $x \in X \Rightarrow e \circ x = x$
- Composition:  $a, b \in G \wedge x \in X \Rightarrow a \circ (b \circ x) = (a \times b) \circ x$

The HOL definition is

$$\begin{aligned} \vdash \text{action } (\circ) g X &\iff \\ \forall x. & \\ x \in X &\Rightarrow \\ (\forall a :: (G). a \circ x \in X) &\wedge e \circ x = x \wedge \\ \forall a b :: (G). a \circ b \circ x &= (a \times b) \circ x \end{aligned}$$

The set  $X$  above is called the *target*. We can picture a target point  $x \in X$  being *acted upon* by the group elements. Alternatively, we say that point  $x$  can *reach* another point  $a \circ x$  for some  $a \in G$ . If  $a \circ x = x$ , we say that the group element  $a$  leaves the point  $x$  *fixed*. This leads to the following:

**Definition 4.** For  $x \in X$ , the set of target points it can reach form its **orbit**.

**Definition 5.** For  $x \in X$ , the set of group elements that leave it fixed form its **stabilizer**.

For example,  $\mathbb{Z}_n^+$  acts on the set of necklaces of length  $n$ , with `cycle` being an action from  $\mathbb{Z}_n^+$  to the necklaces. Each monocoloured necklace always cycles to itself. Thus its orbit consists of itself only, and its stabilizer is all of the group's carrier. For each multicoloured necklace, cycling brings it to another (similar) multicoloured necklace. Since  $|\mathbb{Z}_n^+| = n$ , its orbit contains at most  $n$  reachable points in the target. Generally, more reachable points give a larger orbit, and the corresponding stabilizer is smaller. In the extreme case when the orbit has  $n$  distinct target points, the stabilizer contains just the group identity.

This is a hint that the sizes of orbits and stabilizers may have a relationship — an issue we shall explore.

*HOL Implementation.* The HOL definitions of these concepts pick up multiple parameters, so that, for example, the `reach` relation is not simply a binary notion but has to include explicit parameters for the action and the governing group. Similar extra parameters are required for `orbit` and `stabilizer` definitions:

$$\begin{aligned} \vdash \text{reach } (\circ) g x y &\iff \exists a. a \in G \wedge a \circ x = y \\ \vdash \text{orbit } (\circ) g X x &= \{y \mid y \in X \wedge \text{reach } (\circ) g x y\} \\ \vdash \text{stabilizer } (\circ) g x &= \{a \mid a \in g.\text{carrier} \wedge a \circ x = x\} \end{aligned}$$

For presentational reasons, we shall assume fixed operation  $(\circ)$ , group  $g$  and target set  $X$  in much of what follows, and use the following abbreviations in prose and HOL theorems:

- *orbit*  $x$  for `orbit (o) g X x`, and
- *stabilizer*  $x$  for `stabilizer (o) g x`.

### 5.2 Action Basics

Properties of group actions blend nicely with properties of groups, as shown by these basic results.

**Theorem 12.** *Reachability is an equivalence relation on the target set.*

$$\vdash \text{Group } g \wedge \text{action } (o) \ g \ X \Rightarrow \text{reach } (o) \ g \ \text{equiv\_on } X$$

*Proof.* Let  $x \sim y$  stand for `reach (o) g x y`. By action identity:  $e \circ x = x$ , hence  $x \sim x$ , or `reach` is *reflexive*. If  $a \in G$  moves point  $x$  to  $y$ :  $a \circ x = y$ , then  $a^{-1} \in G$  moves  $y$  to  $x$ :  $a^{-1} \circ y = a^{-1} \circ (a \circ x) = (a^{-1} \times a) \circ x = e \circ x = x$ , hence  $x \sim y \Rightarrow y \sim x$ , or `reach` is *symmetric*. If  $a \circ x = y, b \circ y = z$ , then by action composition:  $(b \times a) \circ x = b \circ (a \circ x) = b \circ y = z$ , hence  $x \sim y \wedge y \sim z \Rightarrow x \sim z$ , or `reach` is *transitive*. Thus `reach` is an equivalence relation.  $\square$

The *orbits* are equivalence classes of `reach`, and they form a partition of the target set  $X$ . Another characterisation of `orbit` using the action mapping  $(\circ)$  is:

$$\vdash \text{Group } g \wedge \text{action } (o) \ g \ X \wedge x \in X \Rightarrow \text{orbit } x = \{a \circ x \mid a \in G\}$$

**Theorem 13.** *The stabilizer of a point in the target set forms a subgroup.*

$$\vdash \text{action } (o) \ g \ X \wedge x \in X \wedge \text{Group } g \Rightarrow \text{StabilizerGroup } (o) \ g \ x \leq g$$

*Proof.* If two elements  $a, b \in G$  fix a point  $x \in X$ , i.e.,  $a \circ x = x$  and  $b \circ x = x$ , then by action composition:  $(a \times b) \circ x = a \circ (b \circ x) = a \circ x = x$ . Therefore, the stabilizer is a closed subset of  $G$ . The identity  $e$  is in the stabilizer by action identity:  $e \circ x = x$ . If  $a$  is in the stabilizer, its inverse  $a^{-1}$  is also in the stabilizer since  $a^{-1} \circ x = a^{-1} \circ (a \circ x) = (a^{-1} \times a) \circ x = e \circ x = x$ . Hence the stabilizer is a subgroup.  $\square$

### 5.3 Orbit-Stabilizer Theorem

Consider a point  $x \in X$ . Its orbit is the set of points reachable through the action of all group elements  $a \in G$ . If all action points  $a \circ x$  are distinct, only  $e \circ x = x$  fixes  $x$ , hence its stabilizer consists of the group identity  $e$  only, the smallest possible subgroup. For example, let  $G = \{a, b, c, d, e, f\}$  with  $e$  being the identity, and  $X = \{x, y, z, \dots\}$ . An example of such a group action is shown in Figure 3.

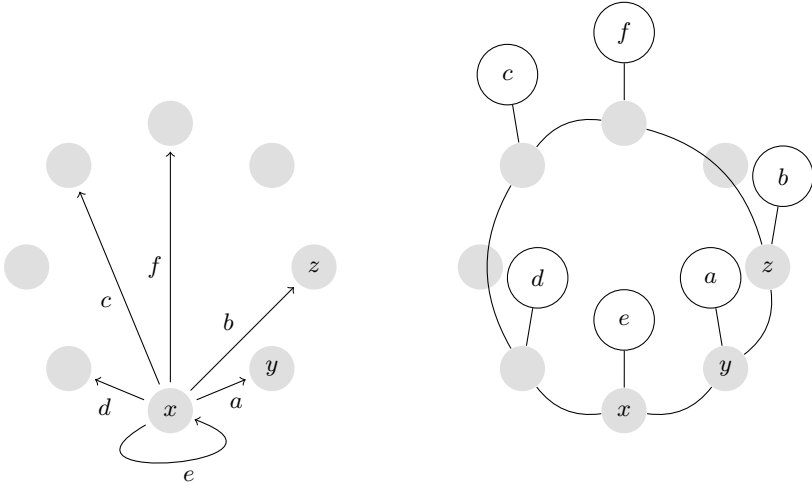
What happens if not all action points are distinct? This is interesting:

**Theorem 14.** *If action points coincide:  $a \circ x = b \circ x$ , the quotient  $a^{-1} \times b$  lies in the stabilizer of  $x$ .*

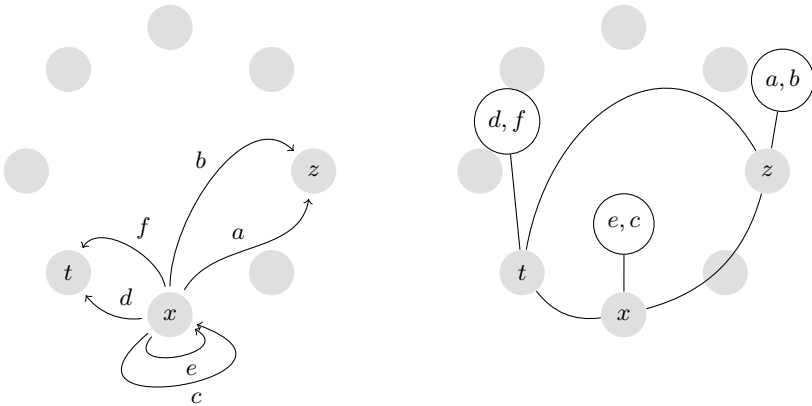
$$\vdash \text{Group } g \wedge \text{action } (o) \ g \ X \wedge x \in X \Rightarrow \forall a \ b :: (G). \ a \circ x = b \circ x \iff a^{-1} \times b \in \text{stabilizer } x$$



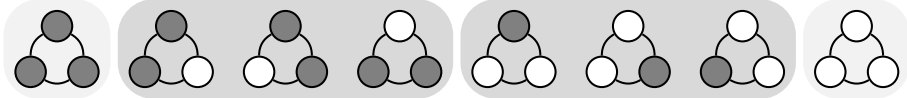
*Proof.* From left to right, apply the action  $a^{-1}$  to both sides of  $a \circ x = b \circ x$ . Thus  $x = (a^{-1} \times b) \circ x$ . From right to left, we have  $(a^{-1} \times b) \circ x = x$ . Apply the action  $a$  to both sides of this equation, deriving  $b \circ x = a \circ x$  as required.  $\square$



**Fig. 3.** If a group action, shown on the left, maps a point  $x \in X$  to distinct reachable points, each reachable point corresponds to only one element in  $G$ . These reachable points can be joined together by arcs, shown on the right, giving the orbit  $x$ . The “balloon” over each  $y \in orbit\ x$  contains the group element which acts on  $x$  to reach  $y$ . Note the balloon over  $x$  is stabilizer  $x$ , in this case just  $\{e\}$ , corresponds to the self-loop over  $x$  on the left.



**Fig. 4.** If stabilizer  $x$ , shown on the left as self-loops over  $x$ , has two elements  $\{e, c\}$ , then every  $z \in orbit\ x$  is reachable by two group elements:  $z = a \circ x = a \circ (c \circ x) = (a \times c) \circ x = b \circ x$  where  $b = a \times c$ , and  $b \neq a$  since  $c \neq e$ . On the right are shown the two group elements for every point in orbit  $x$  inside its “balloon”. The balloon over  $x$  is stabilizer  $x$ ; the other balloons are cosets of stabilizer  $x$ .



**Fig. 5.** Orbits of necklaces with 3 beads in 2 colours under cycle action by  $\mathbb{Z}_3^+$  are shown as background round rectangles. Light gray orbits of monocoloured ones always have size = 1. Dark gray orbits of multicoloured ones always have size > 1. By Orbit-Stabilizer theorem, orbit size divides  $|\mathbb{Z}_3^+| = 3$ , hence multicoloured orbit size = 3.

Furthermore  $a^{-1} \times b$  is some  $c \in G$  by closure property of the group. By uniqueness of group inverses,  $c \neq e$  if  $a \neq b$ . Hence for distinct  $a, b$  with  $a \circ x = z = b \circ x$ , there is an element  $c \neq e$  and  $c \in \text{stabilizer } x$ . Note that  $a^{-1} \times b = c$  implies  $b = a \times c$ . So if, for example (Figure 4),  $\text{stabilizer } x$  is indeed just  $\{e, c\}$ , then the set of group elements enabling  $x$  to reach  $z$ , i.e.  $\{a, b\} = \{a \times e, a \times c\} = a \{e, c\}$ , is a coset of  $\text{stabilizer } x$ .

Similar reasoning shows that, for any point  $y \in \text{orbit } x$ , the set of group elements  $\{a \in G \mid a \circ x = y\}$  that enables  $x$  to reach  $y$  will be a coset of  $\text{stabilizer } x$ . In HOL, this is expressed as:

**Theorem 15.** *The set of group elements enabling  $x$  to reach point  $y \in \text{orbit } x$  is a coset of  $\text{stabilizer } x$ .*

$$\begin{aligned} &\vdash \text{Group } g \wedge \text{action } (\circ) \ g \ X \wedge x \in X \wedge y \in \text{orbit } x \Rightarrow \\ &\quad \{a \mid a \in G \wedge a \circ x = y\} = \\ &\quad \text{coset } g \ (\text{stabilizer } x) \ (\text{actionElement } (\circ) \ g \ x \ y) \end{aligned}$$

where  $(\text{actionElement } (\circ) \ g \ x \ y)$  is a group element that acts on  $x$ , generating  $y$ .

Such an element exists when  $x$  and  $y$  are in the same orbit, as assumed. Since the choice made by  $\text{actionElement}$  may not be drawing on a singleton set, this association of a point  $y \in \text{orbit } x$  with a coset is only meaningful if the coset is independent of this choice. This follows from a standard result about subgroup cosets:

**Theorem 16.** *Two cosets of a subgroup are equal when it has the quotient of their generating elements.*

$$\begin{aligned} &\vdash \text{Group } g \wedge h \leq g \Rightarrow \\ &\quad \forall b \ a :: (G). \text{coset } g \ H \ b = \text{coset } g \ H \ a \iff a^{-1} \times b \in H \end{aligned}$$

*Proof.* For the if-part ( $\Rightarrow$ ), since  $b \in bH$ ,  $bH = aH$  implies there is a  $c \in H$  such that  $b = a \times c$ . Solving for  $c$  in a group:  $c = a^{-1} \times b \in H$ . For the only-if part ( $\Leftarrow$ ), since  $(a^{-1} \times b) \in H$ , so for any  $c \in H$ ,  $(a^{-1} \times b) \times c$  equals to some  $d \in H$ , by closure property of a subgroup. Now  $(a^{-1} \times b) \times c = d$  implies  $b \times c = a \times d$ , for any  $c \in H$ . This shows  $bH \subseteq aH$ . Repeating the same argument with  $b^{-1} \times a = (a^{-1} \times b)^{-1} \in H$ , as a subgroup includes all inverses, gives  $aH \subseteq bH$ . Thus  $bH = aH$ .  $\square$

This matching condition is used to prove the association of stabilizer cosets to orbit points (Theorem 15). Together with the matching condition of reachable points (Theorem 14), both are crucial in establishing:

**Theorem 17.** *The points of  $x$ 's orbit are in bijection with the cosets of  $x$ 's stabilizer.*

$$\begin{aligned} &\vdash \text{Group } g \wedge \text{action } (\circ) g X \wedge x \in X \Rightarrow \\ &\quad \text{BIJ } (\lambda z. \text{coset } g (\text{stabilizer } x) (\text{actionElement } (\circ) g x z)) \\ &\quad (\text{orbit } x) \{ \text{coset } g (\text{stabilizer } x) a \mid a \mid a \in G \} \end{aligned}$$

The last set comprehension is a special form marking  $a$  as the only variable that varies in the leftmost expression,  $\text{coset } g (\text{stabilizer } x) a$ . This bijection provides the key for:

**Theorem 18.** *Orbit-Stabilizer Theorem.*

$$\begin{aligned} &\vdash \text{FiniteGroup } g \wedge \text{action } (\circ) g X \wedge x \in X \wedge \text{FINITE } X \Rightarrow \\ &\quad |G| = |\text{orbit } x| \times |\text{stabilizer } x| \end{aligned}$$

*Proof.* There are  $|\text{orbit } x|$  points in  $x$ 's orbit. Each point is associated with a coset of *stabilizer*  $x$ . Since *stabilizer*  $x$  is a subgroup (Theorem 13), each coset is the same size as *stabilizer*  $x$  (Theorem 11). The cosets form a partition of the carrier set  $G$  (Theorem 9), which is counted by the bijection of Theorem 17:

$$|G| = \sum_{a \in G} |a(\text{stabilizer } x)| = |\text{orbit } x| |\text{stabilizer } x|$$

□

## 5.4 Applying Action to Necklaces

The Orbit-Stabilizer theorem is the key to classifying necklace orbits, especially when the necklace length is prime. First we identify the action:

**Theorem 19.** *cycle is an action from  $\mathbb{Z}_n^+$  to the set of necklaces:*

$$\vdash 0 < n \wedge 0 < a \Rightarrow \text{action cycle } \mathbb{Z}_n^+ (\text{necklace } n a)$$

*Proof.* For necklace  $\ell \in \text{necklace } n a$ ,  $|\ell| = n$ . Each element  $k \in \mathbb{Z}_n^+$ , i.e.  $0 \leq k < n$ , maps a necklace  $\ell$  to the cycle result:  $\text{cycle } k \ell$ , i.e. cycling of the necklace by  $k$  beads. Recall these earlier results about cycle (Section 2.2):

$$\vdash \ell \in \text{necklace } n a \Rightarrow \forall k. \text{cycle } k \ell \in \text{necklace } n a$$

$$\vdash \text{cycle } 0 \ell = \ell$$

$$\vdash \ell \neq [] \Rightarrow \text{cycle } x (\text{cycle } y \ell) = \text{cycle } ((x + y) \bmod |\ell|) \ell$$

The first shows  $\text{cycle}$  is closed for necklaces. The second shows  $\text{cycle}$  has an identity. The third shows  $\text{cycle}$  composes under modulo  $n$  addition. Hence  $\text{cycle}$  is an action from the group  $\mathbb{Z}_n^+$ . □

Since length and colours are invariants for  $\text{cycle}$  (Section 2.2), a multicoloured necklace cannot be cycled to a monocoloured necklace. This shows  $\text{cycle}$  is also closed for those sets respectively:

$$\vdash 0 < n \wedge 0 < a \Rightarrow \text{action cycle } \mathbb{Z}_n^+ (\text{monocoloured } n a)$$

$$\vdash 0 < n \wedge 0 < a \Rightarrow \text{action cycle } \mathbb{Z}_n^+ (\text{multicoloured } n a)$$

The classification of orbits for necklaces is simple:

**Theorem 20.** *Only monocoloured necklaces have orbit size equal to 1.*

$$\begin{aligned} \vdash 0 < n \wedge 0 < a \wedge \ell \in \text{monocoloured } n \ a \Rightarrow \\ \quad |\text{orbit cycle } \mathbb{Z}_n^+ \text{ (monocoloured } n \ a) \ \ell| = 1 \\ \vdash 0 < n \wedge 0 < a \wedge \ell \in \text{multicoloured } n \ a \Rightarrow \\ \quad |\text{orbit cycle } \mathbb{Z}_n^+ \text{ (multicoloured } n \ a) \ \ell| \neq 1 \end{aligned}$$

*Proof.* Only a monocoloured necklace  $\ell$  has  $\text{cycle } 1 \ \ell = \ell$  (Theorem 2), *i.e.* for all multiples  $k$ ,  $\text{cycle } k \ \ell = \ell$  by CYCLE\_ADD. Hence only such orbit collapses to a singleton, with cardinality 1.  $\square$

**Theorem 21.** *For multicoloured necklaces of length  $p$ , a prime, the orbit size of each necklace equals  $p$ .*

$$\vdash \text{prime } p \wedge 0 < a \wedge \ell \in \text{multicoloured } p \ a \Rightarrow \\ \quad |\text{orbit cycle } \mathbb{Z}_p^+ \text{ (multicoloured } p \ a) \ \ell| = p$$

*Proof.* When the necklace length is prime  $p$ , the action group is  $\mathbb{Z}_p^+$ . By the Orbit-Stabilizer theorem (Theorem 17):  $|\text{orbit } \ell| \times |\text{stabilizer } \ell| = |\mathbb{Z}_p^+| = p$  for any necklace  $\ell$ . A prime  $p$  has only trivial factorization:  $p = 1 \times p = p \times 1$ . By Theorem 20, the orbit of a multicoloured necklace is not a singleton, so its size must be  $p$ .  $\square$

Recall that reach is an equivalence relation (Theorem 12). Orbits are the equivalence classes of reach, so they form a partition of the target set. From Theorem 21, the target is (multicoloured  $p \ a$ ) with size  $(a^p - a)$  (Section 2.1). Theorem 21 also specifies an equal-size partition, giving the visual image of division (Figure 5). Hence,  $p$  divides  $a^p - a$ , which in modulo  $p$  is Fermat’s Little Theorem:

$$\vdash \text{prime } p \Rightarrow a^p \equiv a \pmod{p}$$

## 6 Group Theory applied to the Number-Theoretic Proof

Having applied group theory to the necklace proof, it is interesting to try the “same trick” with the number-theoretic proof. The subsequent results are not novel, but allow a fuller comparison of approaches when we conclude.

It is straightforward to recast the number-theoretic proof of Fermat’s Little Theorem (Section 3) in the context of finite Abelian groups, the structure that naturally mimics prime modulo multiplication. The factor rearrangement and cancellation are direct consequences of commutativity and cancellation laws in Abelian groups. However, this is not very illuminating, and unnecessarily restrictive, as the group-theoretic version of Fermat’s Little Theorem holds for all finite groups (not just the Abelian ones):

$$\vdash \text{FiniteGroup } g \wedge a \in G \Rightarrow a^{|\text{G}|} = e$$

Assuming this result (which will be proved later, see Theorem 22), to derive Fermat’s Little Theorem it is sufficient to show that prime modulo multiplication, *i.e.*  $\mathbb{Z}_p^*$  for prime  $p$ , does indeed form a group — with  $|\mathbb{Z}_p^*| = |\{1 \dots p-1\}| = p-1$ , and the multiplicative identity is 1. Critically, we need to show that any  $x \in \{1 \dots p-1\}$  has a multiplicative inverse in  $\mathbb{Z}_p^*$ . This can be done by appeal to Bézout’s identity, a property of gcd already used in Theorem 1:

$$\vdash x \neq 0 \Rightarrow \exists k \ q. \ k \times x = q \times p + \text{gcd } p \ x$$

With  $p$  a prime and  $0 < x < p$ ,  $\text{gcd } p \ x = 1$ . Taking modulo  $p$  on both sides of the equation, the right-hand side becomes 1, and the  $k$  on the left gives  $k \pmod{p}$  as the multiplicative inverse of  $x$  in  $\mathbb{Z}_p^*$ .

### 6.1 Euler’s Generalization

When the modulo  $n$  is not a prime, the non-zeroes of  $\mathbb{Z}_n$  do not form a multiplicative group; e.g. to find the multiplicative inverse for 2 in  $\mathbb{Z}_6$  would require solving  $2x = 6y+1$ , which is impossible by parity. However, Euler observed that the Bézout’s identity of the preceding section actually guarantees a multiplicative inverse for  $x < n$  whenever  $\text{gcd } n \ x = 1$ , i.e.  $x$  is co-prime to  $n$ :

$$\begin{aligned} \vdash 1 < n \wedge 0 < x \wedge x < n \wedge \text{coprime } n \ x &\Rightarrow \\ \exists y. 0 < y \wedge y < n \wedge \text{coprime } n \ y \wedge y \times x \text{ mod } n &= 1 \end{aligned}$$

This is then the basis for a group, whose carrier is  $\mathbb{Z}_n^*$  — the set of elements of  $\mathbb{Z}_n$  with multiplicative inverses. The cardinality of this set is known as its *totient*, denoted by  $\varphi(n)$ :

$$\begin{aligned} \vdash \mathbb{Z}_n^* &= \{x \mid 0 < x \wedge x < n \wedge \text{coprime } n \ x\} \\ \vdash \varphi(n) &= |\mathbb{Z}_n^*| \end{aligned}$$

Euler’s generalisation of Fermat’s Little Theorem follows:

$$\vdash 1 < n \wedge 0 < a \wedge a < n \wedge \text{coprime } n \ a \Rightarrow a^{\varphi(n)} \equiv 1 \pmod{n}$$

We shall now prove the group-theoretic version of Fermat’s Little Theorem for any finite group, *via* the generated subgroup of its elements.

### 6.2 Generated Subgroups

Let  $a$  be a group element. Consider the sequence  $a, a^2, a^3, \dots$ . If the group is finite, there must eventually be a repetition in this sequence. Assume  $m < n$  and  $a^m = a^n$ , then we can use left-cancellation to remove the common factor  $a^m$ , giving us

$$\vdash \text{Group } g \wedge a \in G \wedge m < n \wedge a^m = a^n \Rightarrow a^{n-m} = e$$

**Definition 6.** *Call the least non-zero exponent that maps an element back to the identity, its **order**:*

$$\vdash \text{order } g \ a = \text{LEAST } k. 0 < k \wedge a^k = e$$

The preceding argument shows that **order** exists for finite group elements, and it satisfies:

$$\begin{aligned} \vdash \text{FiniteGroup } g \wedge a \in G &\Rightarrow 0 < \text{order } g \ a \wedge a^{\text{order } g \ a} = e \\ \vdash \text{FiniteGroup } g \wedge a \in G &\Rightarrow a^{-1} = a^{\text{order } g \ a - 1} \end{aligned}$$

By properties of group exponentiation (Section 4), the powers of an element  $a \in G$  form a subgroup: Generated  $g a$ , also written as  $\langle a \rangle$ . This subgroup is related to order by:

$$\vdash \text{FiniteGroup } g \wedge a \in G \Rightarrow |(\text{Generated } g a).\text{carrier}| = \text{order } g a$$

This result can be deduced by the LEAST property of order, and provides the key for:

**Theorem 22.** *Fermat’s Little Theorem for finite groups.*

$$\vdash \text{FiniteGroup } g \wedge a \in G \Rightarrow a^{|\text{G}|} = e$$

*Proof.* Consider  $\langle a \rangle$ , the generated subgroup of  $a \in G$ , with  $|\langle a \rangle| = \text{order } g a$ , and  $a^{\text{order } g a} = e$ . Since  $\langle a \rangle$  is a subgroup of  $g$ , there is a  $k$  such that  $|G| = \text{order } g a \times k$  by Lagrange’s Theorem. Hence,

$$a^{|\text{G}|} = a^{\text{order } g a \times k} = (a^{\text{order } g a})^k = e^k = e$$

as required. □

**Table 1.** Line counts for theories developing each approach. The filename is that of the linearised script in the `bitbucket.org` repository.

Type of Proof	Approach (Section ref.)	Filename	Total
Combinatorial	Direct <i>via</i> cycles (2)	AllFLTnecklaceScript.sml	824
	Group <i>via</i> action (5)	AllFLTactionScript.sml	1387
Number-theoretic	Direct <i>via</i> modulo arithmetic (3)	AllFLTnumberScript.sml	473
	Group <i>via</i> generated subgroups (6)	AllFLTgroupScript.sml	839
	Euler <i>via</i> generated subgroups (6)	AllFLTeulerScript.sml	871

## 7 Conclusion

Fermat’s Little Theorem is a very basic and well-known result in number theory. Having attempted its proof in a slew of different styles, we now attempt to draw some lessons.

For each proof discussed, we have linearised our various script files into one script containing just the lemmas required for that particular effort. Table 1 includes total line counts for each file. The verdict is clear: the basic number-theoretic approach is much better in terms of overall lines-of-code. However, these results can only be suggestive: our HOL scripts may not be the optimal expression of the various proof strategies, and our own skills may not be uniform across the numerical and algebraic domains.

Additionally, HOL4’s features make some proofs easier to automate, and some goals easier to express. Certainly, we believe that our naive approach to group theory makes the numbers for the group-theoretic proofs worse than they might be, and the Orbit-Stabiliser theorem is arguably a steeper requirement than Theorem 22. Sub-types,

perhaps best exemplified by their use in Coq (though also approximated and used for group theory in HOL4 by Hurd [8]), would be an obvious way to approach this issue. Isabelle’s axiomatic type-classes and locales have also been used to provide appealing mechanisations of abstract algebra. It would be interesting to see what these other systems made of the directly combinatorial necklace proof, and of the group-theoretic version of the same.

**Future Work.** The HOL4 source code provides an example of proving Fermat’s Little Theorem using the Binomial Theorem.<sup>5</sup> The proof is by induction, and thus rather different from the proofs in this paper. Indeed, Fermat’s Little Theorem and the Binomial Theorem are crucial concepts in Agrawal *et al.*’s famous result [1] that primality testing can be done in polynomial time. A mechanisation of the AKS algorithm is certainly an appealing prospect.

**Final Verdict.** Our results show that we have yet to find the sweet spot when it comes to performing combinatorial proofs in HOL. Our consolation is to have found that attacking the result *via* explicit appeals to group theory gives us two distinct mechanised proofs that are arguably more elegant than their direct analogues. Our mechanisation of the necklace proofs may be the first; we hope it is not the last, and that still more beautiful pearls may be found in this vein.

## References

1. Agrawal, M., Kayal, N., Saxena, N.: PRIMES is in P. *Annals of Mathematics* 160(2), 781–793 (2004)
2. Conrad, K.: Group actions (2008), <http://www.math.uconn.edu/~kconrad/blurbs/grouptheory/gpaction.pdf>
3. Dickson, L.E.: *History of the Theory of Numbers: vol. 1: Divisibility and Primality*. Carnegie Institution of Washington (1919)
4. Golomb, S.W.: Combinatorial proof of Fermat’s “Little” Theorem. *The American Mathematical Monthly* 63(10), 718 (1956)
5. Gunter, E.L.: *Doing algebra in simple type theory*. Technical Report MS-CIS-89-38, Department of Computer and Information Science, Moore School of Engineering, University of Pennsylvania (June 1989)
6. Harrison, J.: *HOL Light Tutorial (for version 2.20)*. Intel JF1-13, Section 18.2: Fermat’s Little Theorem (2011)
7. Holt, B.V., Evans, T.J.: A group action proof of Fermat’s Little Theorem, <http://arxiv.org/abs/math/0508396>
8. Hurd, J.: Predicate Subtyping with Predicate Sets. In: Boulton, R.J., Jackson, P.B. (eds.) *TPHOLs 2001*. LNCS, vol. 2152, pp. 265–280. Springer, Heidelberg (2001)
9. Hurd, J., Gordon, M., Fox, A.: Formalized elliptic curve cryptography. In: *High Confidence Software and Systems: HCSS 2006* (April 2006)
10. Oostdijk, M.: Library pocklington.fermat, <http://coq.inria.fr/pylons/contribs/files/Pocklington.fermat.html>

<sup>5</sup> This proof is by Laurent Théry, and is apparently itself a translation of a Coq proof by J. C. Almeida.

11. Rouse, J.: Combinatorial proofs of congruences. Master's thesis, Harvey Mudd College (2003)
12. Russinoff, D.: ACL2 Version 3.2 source (2007), [books/quadratic-reciprocityfermat.lisp](#)
13. Slind, K., Norrish, M.: A Brief Overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
14. Smyth, C.J.: A coloring proof of a generalisation of Fermat's Little Theorem. *The American Mathematical Monthly* 93(6), 469–471 (1986)
15. Wikipedia: Proofs of Fermat's Little Theorem, [http://en.wikipedia.org/wiki/Proofs\\_of\\_Fermat's\\_little\\_theorem](http://en.wikipedia.org/wiki/Proofs_of_Fermat's_little_theorem)



# Compact Proof Certificates for Linear Logic

Kaustuv Chaudhuri

INRIA, France

<http://kaustuv.chaudhuri.info>

**Abstract.** Linear logic is increasingly being used as a tool for communicating reasoning agents in domains such as authorization, access control, electronic voting, etc., where proof certificates represent evidence that must be verified by proof consumers as part of higher protocols. Controlling the size of these certificates is critical. We assume that the proof consumer is allowed to do some search to reconstruct details of the full proof that are omitted from the certificates. Because the decision problem for linear logic is unsolvable, the certificate must contain at least enough information to bound the search: we show how to use the sequence of contractions in the sequent proof for this bound. The remaining content of the proof, in particular the information about resource divisions, can then be omitted from the certificate. We also describe a technique for giving a variable amount of additional search hints to the proof consumer to limit its non-determinism.

## 1 Introduction

A *proof certificate* is a way for a *proof producer* to convey certainty about a theorem to a *proof consumer*. It is the embodiment of a compromise between the size of the certificate (which directly correlates to the difficulty of processing, transmitting, and storing the certificate) and the complexity of checking (and hence trusting) it. A fully detailed proof can be very large, but it might be checkable by a simple and trustworthy checker (the so-called *De Bruijn Criterion*). On the other extreme, the certificate might just record a “yes”, and the consumer must reconstruct the entire proof. This paper explores some of the spectrum between these two extremes and provides some guidelines for producing certificates that have a *tunable* amount of detail. Of particular interest is the kind of certificate where the level of detail can be modified by intermediaries between the ultimate producer and consumer, an idea that is at the heart of the “marketplace of proofs” concept [16].

To be simple and concrete, this paper considers this question for classical linear logic, which is undecidable even in the propositional fragment [15]. The ideas readily extend to the first-order and to intuitionistic and classical logic, where many of the issues are simpler. The underlying proof system will be a *sequent calculus*. Sequent calculi are ideally suited for proof-search in many logics for at least two important reasons: first, the subformula property, which is the *sine qua non* of automation; and second, polarity and focusing, wherein the

ordinary sequent rules coalesce into synthetic “macro” forms to make larger logical steps without sacrificing completeness [1]. The careful use of focusing enables a general search strategy to implement a wide variety of operational strategies directly [8,6].

The sequent calculus is nevertheless not an ideal certificate format. It is a simple matter to build a syntax for fully detailed sequent proofs. If the proof is of an important mathematical result, then such detailed proofs can perhaps be tolerated as they are unlikely to be consumed often. Commonly, however, automatically generated proofs are used in domains where the proofs are intended to convince the consumer of some semantic property of digital artefacts, such as conformance to security policies. Probably the best example is proof-carrying code (PCC) [18]. Certificates in such domains are meta-information and generally considered to be overhead. For PCC, the standard technique for reducing the overhead is to specify the search semantics for the consumer (*e.g.*, by means of a logic programming language) and then to record the choices needed to guide the consumer in the form of oracle strings. While this may be a good engineering solution for the specific problem of PCC, oracle strings are an unsatisfactory proof certificate format in general. The strings must match the operational semantics of the consumer, for one, which limits the portability and maintainability of proofs. They are also denotationally opaque.

An alternative to the oracles approach—the one used in this paper—is to *elide* some of the details in the proof if they can be *dependably* reconstructed by the consumer. This is to say that the elision must be such that reconstructing the full sequent proof always remains at least decidable, and preferably feasible and predictable. A good example of this approach is the Dedukti system [2] where purely computational steps in a proof are elided because the consumer is equipped with a general rewrite engine. However, we want more than just the ability to omit certain classes of sub-proofs: we want the level of detail in a proof to be fully variable. Linearity also brings its own problems to such approaches: the key issue is that linear proofs *consume* resources<sup>1</sup> so an omitted proof would consume an unknown amount of resources. Reconstructing the resource divisions is the well known *resource management problem* [10,11], which is unsolvable in general as it is the source of the undecidability of linear logic.

What details are essential in a sequent proof? Every logical rule introduces some connective, so applying the rule from the conclusion to the premises *consumes* the principal formula. Of the structural rules, the only rule that strictly increases the complexity of a sequent is contraction. The contraction-free fragment of the sequent calculus is manifestly decidable, assuming the subformula relation is well-founded and no rule has infinitely many premises – reasonable assumptions for logics where automated reasoning is feasible. The proof-theoretic impact of restricting or removing the contraction rules is a long studied field (see, *e.g.*, [9]), but the following deceptively simple observation seems to be either missing or vanishingly unpopular in the literature on proof objects: *the sequence of contractions in a sequent proof is a sufficient certificate.*

---

<sup>1</sup> A *resource* is a hypothesis that can disappear upon use.

One reason why this observation might be rare is that it is unclear how to isolate the contractions from the other rules in the sequent calculus. Other proof calculi, such as the *calculus of structures* [19,7], permit inferences in any formula context and thus allow the contraction rule to permute freely. Separating the contractions from an arbitrary proof is routine in such formalisms and generally forms the basis of much of their meta-theory. From the perspective of proof search, this extra permutative freedom is actually detrimental, for the proof search trees are much more branching. But, once a proof is built, it can be freely reorganized. The record of contractions is generally much smaller than the full proof—which needs to record the contractions anyhow—because it lacks all the logical content.

Now, while the contraction rules in the sequent calculus obviously do not permute, what is important is not the ability to permute but a mechanism to *pre-compute* the contractions. This pre-computed information about just the contractions can then be used to control their application in a general search strategy, so they form a suitable proof certificate. To extract this information, we require a mechanism for uniquely indexing every subformula in a sequent. This is not particularly hard: we merely give a unique name to every subformula path in the sequent and perform some additional book-keeping to ensure that the names remain unique through applications of sequent rules. The contractions then manifest as subformula paths that can potentially be duplicated; this information can then be used as a *bound* on search, wherein every application of contraction “consumes” one of the copies of the replicable subformula paths.

Bounding the contractions in this manner makes proof search (and hence reconstruction in the consumer) decidable, but it does not necessarily make it feasible. As already mentioned, a good certificate format should allow a variable level of detail to control the non-determinism in the consumer. It turns out that, carefully done, the indexing mechanism used for the contractions can then be exploited in another important manner: the tree of names of the principal formulas in the sequent proof is also a suitable certificate format. Moreover, this tree can serve as a skeleton from which to hang the information about contractions. This is the main technical contribution of this paper.

We begin by limiting ourselves to focused proofs [18,14]. The essence of focusing is to reduce the subformula relation to one that clarifies the alternation between invertible (“don’t care”) and non-invertible (“don’t know”) choices in search. Only these points of *phase shift* need to be indexable. Of the phase shifts, the most important ones are those that *decide* to focus on a particular formula: the corresponding labels are recorded in the *decision tree* of labels. This crystallizes the common intuition that the essence of a focused proof is the sequence of decisions; the details of the choices *inside* particular phases of focusing are unimportant. In fact, the other non-deterministic choices in a proof, *viz.* the resource distribution and the disjunctive choices, can be recovered directly from this decision tree. Lastly, at any level in the decision tree we can simply replace the sub-tree with a bound on the contractions, which gives us certificates with a variable level of detail.

$$\begin{array}{c}
 \frac{}{\vdash \bar{a}, a} \text{ai} \quad \frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \otimes \quad \frac{}{\vdash \mathbf{1}} \mathbf{1} \quad \frac{\vdash \Gamma, A_i}{\vdash \Gamma, A_1 \oplus A_2} \oplus \quad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \wp B} \wp \quad \frac{\vdash \Gamma}{\vdash \Gamma, \perp} \perp \\
 \frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \& B} \& \quad \frac{}{\vdash \Gamma, \top} \top \quad \frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A} ! \quad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A} ? \quad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A} \text{ct} \quad \frac{\vdash \Gamma}{\vdash \Gamma, ?A} \text{wk}
 \end{array}$$

**Fig. 1.** Rules of LLK. In the  $\oplus$  rule,  $i \in \{1, 2\}$ .

To summarize, our prescription for obtaining compact proof certificates is as follows: (1) start from a focused sequent proof; (2) uniquely name every subformula path in the end-sequent; (3) extract the tree of names for the principal formulas in the decision rules; and (4) replace some of the sub-trees in this tree by a suitable search bound, such as one on the contractions. We begin with an overview of the focused sequent calculus (Sect. 2), then we show how to label the sequents and tame contraction (Sect. 3), and finally we describe how to use the decision trees as determinizing hints for proof reconstruction (Sect. 4).

## 2 Background

We use propositional linear logic in this paper, though the technique extends straightforwardly to the first-order. It also extends to ordinary logic (intuitionistic or classical) where the problems are simpler than the linear case. Formulas (written  $A, B, \dots$ ) have the following grammar.

$$\begin{array}{l}
 A, B, \dots ::= a \mid A \otimes B \mid \mathbf{1} \mid A \oplus B \mid \mathbf{0} \mid !A \\
 \quad \quad \quad \mid \bar{a} \mid A \wp B \mid \perp \mid A \& B \mid \top \mid ?A
 \end{array}$$

Atomic formulas are written using  $a, b, \dots$ , and the negation of  $a$  is written as  $\bar{a}$ . Formulas are in negation-normal form with each vertical column in the above grammar depicting one De Morgan dual pair; we write  $(A)^\perp$  for the dual of  $A$ . The linear sequent calculus, which we call LLK, is usually presented using one-sided sequents of the form  $\vdash \Gamma$ , where  $\Gamma$  is a multi-set of formulas. The rules of this system are in Fig. 1.

Looking at the permutations of rules in LLK, it is easy to see that some rules can always permute because they are invertible (*i.e.*, if the conclusion of the rule is true, then so is the conjunction of its premises), while other (non-invertible) rules permute only in specific circumstances. Andreoli famously showed that these permutation properties can be exploited to define a restricted but complete class of sequent proofs that follow a *focusing* discipline [2]. The essence of the discipline is that the ordinary (unfocused) rules of LLK naturally coalesce into larger clumps of derived rules that abstract over the irrelevant details such as the order of application of invertible rules. Focusing was originally an operational device to control the non-determinism in proof-search in linear logic programming, but it is now seen as a general device for analyzing the structural properties of proof systems, akin to cut-elimination for the sequent calculus. Focused proof systems have been formulated for a number of other logics [8, 14] and proof systems [3, 7] besides its origin in the classical linear sequent calculus.

The standard presentation of focusing for LLK is as follows, in broad outline. The non-atomic formulas of linear logic divide evenly into those that have invertible rules and those that do not; moreover, the two sets of formulas are De Morgan duals. Following general tradition, we call the connectives with invertible logical rules *negative*, and those with non-invertible rules *positive*. The atomic formulas can be placed into either class as long as duality is maintained, but we follow tradition and classify them as positive and their negations as negative. We also adopt the device of *polarization* [12], wherein the changes between positive and negative subformulas is explicitly marked with *shift* connectives ( $\downarrow$  and  $\uparrow$ ). While the choice of a polarized syntax is usually unnecessary for focusing, we will exploit it for our certificates.

Polarized formulas have the following grammar.

$$\begin{aligned} P, Q, \dots & ::= a \mid P \otimes Q \mid \mathbf{1} \mid P \oplus Q \mid \mathbf{0} \mid !N \mid \downarrow N && \text{(positive formulas)} \\ N, M, \dots & ::= \bar{a} \mid N \wp M \mid \perp \mid N \& M \mid \top \mid ?P \mid \uparrow P && \text{(negative formulas)} \end{aligned}$$

We will continue to use  $A, B, \dots$  to refer to formulas of either polarity. The sequents in the focused variant of LLK, which we call LLKF, have one the following two forms.

$$\begin{aligned} \vdash \Gamma; \Delta; [P] &&& \text{(positive sequent with } P \text{ focused)} \\ \vdash \Gamma; \Delta; \Omega &&& \text{(negative sequent with } \Omega \text{ active)} \end{aligned}$$

The contexts  $\Gamma$ ,  $\Delta$  and  $\Omega$  are all multi-sets of the following kinds of formulas.

$$\Gamma ::= \cdot \mid \Gamma, P \qquad \Delta ::= \cdot \mid \Delta, P \mid \Delta, \bar{\pi} \qquad \Omega ::= \cdot \mid \Omega, N$$

Although all three contexts are multi-sets, semantically the context  $\Gamma$  is *unrestricted* (admitting weakening and contraction) while  $\Delta$  and  $\Omega$  are *linear* (admitting neither weakening nor contraction).

Figure 2 contains the full collection of rules of LLKF. The logical rules of LLKF are applied in one of two *phases*. The positive phase consists of rules applied to positive sequents. Each such rule has the focused formula in the positive sequent as the principal formula, and if the operands of the principal connective are also positive then the focus is maintained on them in their corresponding premises. Likewise, the negative phase consists of rules applied to the active context of negative sequents. Mediating the two phases are the *decision rules* [lf] and [uf] where particular formulas are granted focus. In the case that the formula is selected from the unrestricted context  $\Gamma$ , it continues to be present in  $\Gamma$  in the premises, *i.e.*, the rule has a built in contraction. This is in fact the only form of contraction in the calculus. Instead of a structural rule of weakening, rules with no premises are altered to admit any number of unrestricted side formulas.

In order to compare LLKF to LLK, we must first translate between the two syntaxes – unpolarized and polarized.

**Definition 1 (Depolarization).** *Given a polarized formula  $A$ , let  $\lfloor A \rfloor$  stand for that unpolarized formula with all occurrences of  $\downarrow$  and  $\uparrow$  removed from  $A$ . Extend this definition naturally to multi-sets of polarized formulas.*

Positive Phase

$$\frac{}{\vdash \Gamma; \bar{a}; [a]} [\text{fi}] \quad \frac{\vdash \Gamma; \Delta_1; [P] \quad \vdash \Gamma; \Delta_2; [Q]}{\vdash \Gamma; \Delta_1, \Delta_2; [P \otimes Q]} [\otimes] \quad \frac{}{\vdash \Gamma; \cdot; [1]} [1] \quad \frac{\vdash \Gamma; \Delta; [P_i]}{\vdash \Gamma; \Delta; [P_1 \oplus P_2]} [\oplus]$$

$$\frac{\vdash \Gamma; \cdot; N}{\vdash \Gamma; \cdot; [!N]} [!] \quad \frac{\vdash \Gamma; \Delta; N}{\vdash \Gamma; \Delta; [\downarrow N]} [\downarrow]$$

Negative Phase

$$\frac{\vdash \Gamma; \Delta; \bar{a}; \Omega}{\vdash \Gamma; \Delta; \Omega; \bar{a}} [\text{nr}] \quad \frac{\vdash \Gamma; \Delta; \Omega, N, M}{\vdash \Gamma; \Delta; \Omega, N \wp M} [\wp] \quad \frac{\vdash \Gamma; \Delta; \Omega}{\vdash \Gamma; \Delta; \Omega, \perp} [\perp] \quad \frac{\vdash \Gamma; \Delta; \Omega, N \quad \vdash \Gamma; \Delta; \Omega, M}{\vdash \Gamma; \Delta; \Omega, N \& M} [\&]$$

$$\frac{}{\vdash \Gamma; \Delta; \Omega, \top} [\top] \quad \frac{\vdash \Gamma, P; \Delta; \Omega}{\vdash \Gamma; \Delta; \Omega, ?P} [?] \quad \frac{\vdash \Gamma; \Delta, P; \Omega}{\vdash \Gamma; \Delta; \Omega, \uparrow P} [\uparrow]$$

Decision

$$\frac{\vdash \Gamma; \Delta; [P]}{\vdash \Gamma; \Delta, P; \cdot} [\text{lf}] \quad \frac{\vdash \Gamma, P; \Delta; [P]}{\vdash \Gamma, P; \Delta; \cdot} [\text{uf}]$$

**Fig. 2.** Rules of LLKF. In the  $[\oplus]$  rule,  $i \in \{1, 2\}$ .

### Theorem 1 (Soundness and completeness of LLKF w.r.t. LLK)

- If  $\vdash \Gamma; \Delta; [P]$  in LLKF, then  $\vdash ?[\Gamma], [\Delta], [P]$  in LLK.
- If  $\vdash \Gamma; \Delta; \Omega$  in LLKF, then  $\vdash ?[\Gamma], [\Delta], [\Omega]$  in LLK.
- If  $\vdash [\Omega]$  in LLK, then  $\vdash \cdot; \cdot; \Omega$  in LLKF.

*Proof.* There are many ways to prove this theorem. We refer the interested reader to one of the standard approaches [13,17,5].  $\square$

## 3 Labelling Subformulas and Taming Contraction

As already mentioned in the introduction, there is a single rule each in LLK and in LLKF that causes the set of derivations (including open derivations) of a given sequent to be infinite: contraction (ct) in the former, and unrestricted focus ([uf]) in the latter. The [lf] rule moves a positive formula into focus after which its principal connective is consumed, and the [nr] rule moves a negative atom into the linear context from which it can never be selected as a principal formula again. The remaining rules all consume a connective. Therefore, in order to obtain a decidable sub-logic of linear logic for which these proof systems are manifestly decision procedures, it is these rules of contraction that need to be controlled.

To find such a means of controlling contraction, we can look for inspiration at calculi with more permissive permutations in their inference rules. Generally speaking, such calculi tend to be calculi of *deep inference*, wherein there is no strong distinction between sequent and formula, and inference rules can be applied in any subformula context. The system LSF for classical linear logic in the

focused calculus of structures [7] is perhaps the most closely related system to LLKF in this paper. In LSF (like in nearly every proof system in the calculus of structures), the contraction rules permute with all rules (including other contractions). Any LSF proof can therefore be divided into two phases: a bottom part consisting of the contractions, and a top part that is contraction-free. If the system is designed carefully, as LSF is, then this contraction-free sub-calculus admits only finitely many (possibly open) derivations of a given formula, and is therefore decidable.

While it is possible to adopt LSF instead of LLKF to represent proofs, this will require a complicated and not particularly enlightening detour. Instead, we will just transplant the *effect* of permuting and separating the contractions to LLKF proofs. The mechanism we will use is *labelling* the contractions: both the formulas and the inference rules will be modified to admit labels.

**Definition 2.** A label (written  $\alpha, \beta, \dots$ ) is a non-empty word formed over an infinite set of atomic labels (written  $\mathbf{a}, \mathbf{b}, \dots$ ). Two distinct atomic labels  $\mathbf{0}$  and  $\mathbf{1}$  are always assumed to be present. We use  $\mathcal{L}$  to denote the set of all labels,  $\Lambda$  to denote label multi-sets, and  $\alpha\beta$  to denote the label formed by concatenating the labels  $\alpha$  and  $\beta$ .

The labels will be used to index particular subformulas in a derivation in a labelled version of LLKF, which we call L3KF. Intuitively, a label denotes a *path* through the subformula relation, with the formula labelled by  $\alpha\beta$  being a strict subformula of that labelled by  $\alpha$ . Not every subformula needs to be labelled – those subformulas that do not involve any polarity changes or boundary conditions can remain unlabelled. Instead, we affix labels only to the shifts, the exponentials, and the atomic formulas.

**Definition 3.** Action formulas (written  $L$ ) and reaction formulas (written  $R$ ) are given by the following grammar.

$$R ::= a^a \mid !^a N \mid \downarrow^a N \qquad L ::= \bar{a}^a \mid ?^a P \mid \uparrow^a P$$

Labelled polarized formulas *have the following grammar.*

$$\begin{aligned} P, Q, \dots &::= R \mid P \otimes Q \mid \mathbf{1} \mid P \oplus Q \mid \mathbf{0} \\ N, M, \dots &::= L \mid N \wp M \mid \perp \mid N \& M \mid \top \end{aligned}$$

Strictly speaking, for tracking contractions we do not need to label any but the  $?$ -formulas. We choose to label all (re)action subformulas anticipating their use in the next section. For the rest of this paper, unless indicated, we will work solely with labelled polarized formulas. The contexts of L3KF are modifications of those of LLKF to support labelled formulas.

$$\Gamma ::= \cdot \mid \Gamma, \langle \alpha : P \rangle \qquad \Delta ::= \cdot \mid \Delta, \langle \alpha : P \rangle \mid \Delta, \langle \alpha : \bar{a}^a \rangle \qquad \Omega ::= \cdot \mid \Omega, \langle \alpha : N \rangle$$

In addition to these labelled contexts, the focused formula in positive sequents will also be labelled, written as  $[\alpha : P]$ . From any L3KF sequent we can index particular subformulas using the labels.

Positive Phase

$$\begin{array}{c}
 \frac{}{\vdash \Gamma; \langle \alpha : \bar{a}^a \rangle; [\beta : a^b]} [\text{fi}] \quad \frac{\vdash \Gamma_1; \Delta_1; [\alpha : P] \quad \vdash \Gamma_2; \Delta_2; [\alpha : Q]}{\vdash \Gamma_1, \Gamma_2; \Delta_1, \Delta_2; [\alpha : P \otimes Q]} [\otimes] \quad \frac{}{\vdash \Gamma; \cdot; [\alpha : \mathbf{1}]} [\mathbf{1}] \\
 \\
 \frac{\vdash \Gamma; \Delta; [\alpha : P_i]}{\vdash \Gamma; \Delta; [\alpha : P_1 \oplus P_2]} [\oplus] \quad \frac{\vdash \Gamma; \cdot; \langle \alpha a : N \rangle}{\vdash \Gamma; \cdot; [\alpha : !^a N]} [!] \quad \frac{\vdash \Gamma; \Delta; \langle \alpha a : N \rangle}{\vdash \Gamma; \Delta; [\alpha : \downarrow^a N]} [\downarrow]
 \end{array}$$

Negative Phase

$$\begin{array}{c}
 \frac{\vdash \Gamma; \Delta; \langle \alpha : \bar{a}^a \rangle; \Omega}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : \bar{a}^a \rangle} [\text{nr}] \quad \frac{\vdash \Gamma; \Delta; \Omega; \langle \alpha : N \rangle, \langle \alpha : M \rangle}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : N \wp M \rangle} [\wp] \quad \frac{\vdash \Gamma; \Delta; \Omega}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : \perp \rangle} [\perp] \\
 \\
 \frac{\vdash \Gamma; \Delta; \Omega; \langle \alpha : N \rangle \quad \vdash \Gamma; \Delta; \Omega; \langle \alpha : M \rangle}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : N \& M \rangle} [\&] \quad \frac{}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : \top \rangle} [\top] \\
 \\
 \frac{\vdash \Gamma; \langle \alpha a : P \rangle; \Delta; \Omega}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : ?^a P \rangle} [?] \quad \frac{\vdash \Gamma; \Delta; \langle \alpha a : P \rangle; \Omega}{\vdash \Gamma; \Delta; \Omega; \langle \alpha : \uparrow^a P \rangle} [\uparrow]
 \end{array}$$

Decision

$$\frac{\vdash \Gamma; \Delta; [\alpha : P]}{\vdash \Gamma; \Delta; \langle \alpha : P \rangle; \cdot} [\text{if}] \quad \frac{\vdash \Gamma; \langle \alpha 0 : P \rangle; \Delta; [\alpha 1 : P]}{\vdash \Gamma; \langle \alpha : P \rangle; \Delta; \cdot} [\text{uf}]$$

**Fig. 3.** Rules of L3KF. In the  $[\oplus]$  rule,  $i \in \{1, 2\}$ .

**Definition 4 (indexing).** Given an L3KF sequent  $\sigma$  and a label  $\alpha$ , we write  $\sigma(\alpha)$  for an arbitrary (re)action subformula of  $\sigma$  such that:

- $\alpha$  is of the form  $\beta\gamma$  (with  $\gamma$  possibly empty);
- $\beta$  labels some contextual element of  $\sigma$ , i.e., the contexts of  $\sigma$  contain an element of the form  $\langle \beta : A \rangle$ ; and
- $\gamma$  indicates that subformula of  $A$  reached by the trail of atomic labels in  $\gamma$ .

For example, if the sequent  $\sigma$  contains the labelled element  $\langle \alpha : a^b \oplus \downarrow^c(N \wp \bar{c}^d) \rangle$  in a context, then  $\sigma(\alpha b) = a^b$ ,  $\sigma(\alpha c) = N \wp c^d$ , and  $\sigma(\alpha cd) = \bar{c}^d$ . Indexing can be non-deterministic if there are duplicates of labels in an L3KF sequent. We will generally only work with sequents where there are no duplicates, which we call standard sequents.

**Definition 5.** We say that an L3KF sequent  $\sigma$  is standard if for every label  $\alpha$ , there is at most one labelled formula  $A$  such that  $\sigma(\alpha) = A$ .

We assume that all L3KF sequents in the the rest of the paper are standard. We shall design the rules of L3KF in such a way that if the end-sequent is standard, then in every (possibly open) L3KF derivation of that sequent all intermediate sequents are also standard.



The full set of rules of L3KF is shown in Fig. 3. For the rules involving shifts and exponentials, the atomic label on the principal formula is appended to the relevant label in the sequent. The other rules preserve the labels from conclusion to premises. The binary rules  $[\otimes]$  and  $[\&]$  are the only rules that cause a complete duplication of the labels in the side formulas; however, if the conclusion is standard, then each premise of these two rules is individually also standard. For the  $[\wp]$  rule, although the label for the principal formula is duplicated, if the conclusion is standard then each operand of the  $\wp$  will have a disjoint set of atomic labels in its respective subformulas and hence the premise is also standard.

The only rule that differs from the pattern is the  $[\text{uf}]$  rule that appends a new atomic label 0 or 1 to the end of the contextual label of the principal formula. Repeated applications of this rule on the same principal formula will keep one version with a sequence of 0s appended in the unrestricted context, while the remaining copies will be focused on and decomposed. As we intend to control this rule, we will impose a global restriction on the number of 0s that can be appended to a given label: we call this a *contraction bound*.

**Definition 6.** A contraction bound  $\mathcal{B}$  is a total function from  $\mathcal{L}$  to  $\mathbf{N}$  that maps all but a finite set of labels to 0s with the additional property that for every  $\alpha \in \mathcal{L}$ , if  $\mathcal{B}(\alpha) = n > 0$  then  $\mathcal{B}(\alpha 0) = n - 1$ .

**Definition 7.** Given a contraction bound  $\mathcal{B}$ , the system  $\text{L3KF}(\mathcal{B})$  is a proof system consisting of the inference rules of L3KF (Fig. 3) such that all the instances of the  $[\text{uf}]$  rule with principal formula  $\langle \alpha : P \rangle$  have the property that  $\mathcal{B}(\alpha) > 0$ .

*Remark 1.* Any L3KF sequent has only finitely many (possibly open)  $\text{L3KF}(\mathcal{B})$  derivations, as the contraction bounds get stricter with more 0s.  $\square$

Obviously, therefore,  $\text{L3KF}(\mathcal{B})$  for an arbitrary  $\mathcal{B}$  is not complete with respect to LLKF (nor to LLK) because propositional linear logic is undecidable [15]. Yet, for any (possibly open) LLKF derivation we can indeed construct a  $\mathcal{B}$  such that the corresponding labelled form of the LLKF end-sequent is provable in  $\text{L3KF}(\mathcal{B})$ . To make this formal, we compare the LLKF and L3KF systems modulo labelling.

**Definition 8.** Given an L3KF sequent  $\sigma$ , we write  $\text{unl}(\sigma)$  for that LLKF sequent that: replaces all instances of  $\langle \alpha : A \rangle$  in  $\sigma$  by  $A$ , then erases all labels from the (re)action subformulas of  $\sigma$ .

## Theorem 2 (soundness and completeness)

1. For any  $\sigma$  derivable in  $\text{L3KF}(\mathcal{B})$ , the sequent  $\text{unl}(\sigma)$  is derivable in LLKF.
2. For any standard  $\sigma$  for which  $\text{unl}(\sigma)$  is derivable in LLKF, there is a contraction bound  $\mathcal{B}$  such that  $\sigma$  is derivable in  $\text{L3KF}(\mathcal{B})$ .

*Proof (Sketch).* Each case follows by a simple induction.

1. If  $\text{unl}(-)$  is applied to every sequent in every inference rule of  $\text{L3KF}(\mathcal{B})$ , then the result is an inference rule of LLKF.

2. Begin with an empty bound and read the LLKF derivation from conclusion upwards. Whenever  $[\text{uf}]$  is applied in the LLKF derivation, for the corresponding label  $\alpha$  in the L3KF derivation we set  $\mathcal{B}(\alpha) = 1$ , then increment the bounds for every prefix of  $\alpha$  formed by removing 0s from the end. We then perform this instance of  $[\text{uf}]$  in the  $\text{L3KF}(\mathcal{B})$  derivation. For all the other rules, the contraction bound remains untouched and the L3KF rule is the obvious one from Fig. 3.  $\square$

The above completeness theorem is much more general than needed. As we intend the contraction bounds to be used in proof certificates for LLKF derivations, we may give ourselves the freedom to choose a labelling for the end-sequent. In fact, we may choose a most parsimonious simple labelling.

**Definition 9.** *An L3KF sequent is said to be simply labelled if there is at most a single occurrence of every atomic label in the sequent. In other words, no two contextual elements share a non-empty label prefix, and every (re)action subformula has a unique atomic label.*

It is easy to see that a simply labelled sequent is standard. In an implementation, if the LLKF sequents have some canonical universal representation, then this simply labelled form is predictable and so the labelled form of the LLKF end-sequent need not be recorded in the proof certificate. Nevertheless, to be general, we will mention the simply labelled forms in the certificates.

**Definition 10.** *A contraction certificate for an LLKF sequent  $\sigma$  is a pair  $(\tau, \mathcal{B})$  where: (1)  $\tau$  is a simply labelled L3KF sequent with  $\text{unl}(\tau) = \sigma$ ; and (2)  $\mathcal{B}$  is a contraction bound.*

Contraction certificates obviously exist even for unprovable LLKF sequents. Completeness (Thm. 2.2) guarantees that any provable LLKF sequent will have a corresponding contraction bound  $\mathcal{B}$  for which the simply labelled form is provable in  $\text{L3KF}(\mathcal{B})$ . To consume—check—a contraction certificate is equivalent to constructing this  $\text{L3KF}(\mathcal{B})$  proof knowing just the end-sequent and the contraction bound. Now, for any contraction bound  $\mathcal{B}$ , the system  $\text{L3KF}(\mathcal{B})$  is manifestly a decision procedure. After all, L3KF has only finitely many open derivations (Remark 1). Thus, in order to consume a contraction certificate  $(\tau, \mathcal{B})$ , it is sufficient to enumerate all  $\text{L3KF}(\mathcal{B})$  derivations of  $\tau$ , succeeding if any one of them is an  $\text{L3KF}(\mathcal{B})$  proof. The LLKF proof of  $\text{unl}(\tau)$  can be reconstructed from this  $\text{L3KF}(\mathcal{B})$  proof by applying the procedure outlined in the proof of Thm. 2.1.

To represent the contraction certificate, we require no more space than the product of the number of labels in the end-sequent and the number of uses of the contraction rule. This will always be smaller than the full proof (which needs to record the contractions anyhow) because it omits all the logical content of the proof. However, it is an easy exercise to construct a series of problems where the number of required uses of contraction grows exponentially, so in the worst case the contraction certificate will not necessarily improve over the full proof by more than a polynomial factor. In practical uses of proof certificates, however,

the contractions will only be expected to be used for “facts” from the ambient unrestricted context (in other words, the axioms in the theory and the lemmas), which is not so pathological. Indeed, in the very expressive multi-set rewriting fragment of linear logic, the only uses of contraction will be for the (encoding of the) rewrite rules, and there will be exactly as many contractions as steps in the trace. The contraction bounds in the corresponding contraction certificates will be considerably smaller than the full proofs; indeed, the space requirement for the bound will be linear in the length of the trace.

## 4 Determinizing Hints

We can potentially declare success at this point, but it is worth noting that consuming a contraction certificate by enumerating all proofs up to a bound may not be very practical. If the LLKF proof is of a purely MALL formula, then there are no occurrences of  $[uf]$  at all, and hence the contraction bound will be empty. Since the proof certificate records none of the logical rules, the reconstruction of the L3KF( $\mathcal{B}$ ) proof is then at least as computationally expensive as searching for the MALL proof, which is a PSPACE-complete problem [15]. In this section, we will add more information to the proof certificates to make reconstruction more deterministic in exchange for an increase in the size of the certificates. This additional detail in the certificate will be a tunable parameter: with enough detail, the reconstruction should be completely deterministic, but a certificate without any detail should still remain consumable.

To motivate the additions, let us first consider the kinds of information that are recorded in a fully detailed proof. For linear logic, we have the following general non-deterministic choices when searching for an LLKF (or an L3KF) proof.

- Choices between multiple rules for the same principal formula, caused by the  $[\oplus]$  rule, also known as *disjunctive non-determinism*.
- Choices involving splitting the linear context in the  $[\otimes]$  rule, also known as *multiplicative non-determinism*.
- Choices of foci in the  $[lf]$  and  $[uf]$  rules, or the *decision non-determinism*.

(In the first-order case, constructing the existential witnesses is also non-deterministic, which is very similar to the disjunctive case.) None of the other choices matter for focused search. In particular, the order of application of the rules in the negative phase is immaterial. Every unfinished premise of a negative LLKF (or L3KF( $\mathcal{B}$ )) sequent will be *neutral* (i.e., of the form  $\vdash \Gamma; \Delta; \cdot$ ). Regardless of the order of application of the negative rules there will always be the same multi-set of neutral premises of a negative sequent.

In order to determinize proof reconstruction, the certificate will have to record these non-deterministic choices. For disjunctive choices, one of the operands of a  $\oplus$  formula disappears from the sequent. Recall that in a standard L3KF sequent, every (re)action subformula is indexed by a unique label. Therefore, for the operand of the  $\oplus$  rule that disappears, so will all the indexes associated to the subformulas of that lost operand. Since the positive phase must (eventually)

finish<sup>2</sup> by one of  $[fi]$ ,  $[!]$  or  $[\downarrow]$ , it follows that exactly one of the topmost atomic labels in the focused  $\oplus$  formula will eventually be mentioned in the derivation above, so the choice made in the  $\oplus$  rule can be deduced from the indexes in the sequents higher in the proof.

For the multiplicative choices, we can use the input-output interpretation of the linear context [10,11]. Briefly, the entire linear context is sent to the left premise of an instance of  $[\otimes]$ ; this premise consumes as much of the context as it needs and sends the rest to the right premise, which in turn sends its unconsumed portion “down” the proof. The proof of the end-sequent is accepted if it is able to consume the entire linear context. The input-output interpretation thus determinizes the multiplicative non-determinism, *i.e.*, backtracking over different ways of splitting the context is unnecessary. It is nevertheless not complete: it forces a sequence between different multiplicative—and semantically concurrent—branches of the proof, and so an adversarial problem can be constructed for which committing to, say, the left premise before the right will lead to infinitely deep proofs. This incompleteness is not an issue for us, however, as the contraction bound makes all derivations finite. No matter which multiplicative branch is scheduled first, the search procedure on that branch must terminate within the bound. Because this technique is well known and standard, we omit a more detailed and formal description in this paper.

This leaves only the focusing decisions. An obvious way to record these is to just extract the tree of decision rules—which we will call the *decision tree* (to be formalized presently)—from the  $L3KF(\mathcal{B})$  proof. Because every sequent in the derivation is standard, the contextual label of the principal formula in the decision rule is unambiguous. Hence, the decision tree can be straightforwardly built using these labels for the internal nodes. Still, this representation is not wholly satisfactory: the decision tree is, in the worst case, a constant fraction of the size of the entire  $L3KF(\mathcal{B})$  proof.<sup>3</sup> To save space, proof certificates must be allowed to omit portions of the full tree.

Now, the complete decision tree already contains a record of all the decision rules required to build a proof, and hence an additional bound such as one on contractions is redundant. But, if we omit portions of the tree, it does become important to record the contraction bounds so that reconstruction remains decidable. A single contraction bound for the entire proof can certainly be recorded in the certificate anyway, but we can avoid the redundancy with the (recorded portion of the) decision tree by a simple trick. For every unrecorded suffix of the decision tree, we compute locally the contraction bound of the corresponding sub-proof and make it the leaf of the tree. In other words, the certificate would contain a prefix of the decision tree, with contraction bounds at the leaves that correspond to omitted sub-proofs.

It should be intuitively obvious that reconstruction using this representation is decidable, as every sub-proof is built either deterministically from the record

---

<sup>2</sup> Reading, as usual, in the direction of conclusion to premises.

<sup>3</sup> In practice, of course, this fraction will tend to be small because focusing already eliminates much of the noise in LLK proofs.

of focusing steps in the tree or by bounded search using the contraction bounds. It is also clear that proof reconstruction will get increasingly deterministic as more of the decision tree is recorded. The level of detail in the certificate thus becomes a tunable parameter that can be tailored to particular needs or even negotiated between the producer and the consumer.

Let us now crystallize these intuitions with formal definitions.

**Definition 11.** A decision tree  $\mathcal{D}$  is a tree where each node: (1) contains a pair  $\langle \alpha, \Lambda \rangle$  where  $\alpha$  is a label and  $\Lambda$  is a set of atomic labels, and has a finite number (possibly zero) of children; or (2) contains a contraction bound (Defn. 10) and no children. A decision tree is full if it contains no nodes with contraction bounds.

The pairs  $\langle \alpha, \Lambda \rangle$  are interpreted as follows:  $\alpha$  is the contextual label of the principal formula of a corresponding decision rule ( $[lf]$  or  $[uf]$ ), and  $\Lambda$  represents the labels of all the reaction formulas—*i.e.*, the labels of the principal formulas of the  $[fi]$ ,  $[!]$  and  $[\downarrow]$  rules—at the boundaries of the positive phase that immediately follows (reading from conclusion upwards) the decision rule. Any disjunctive choices made in the positive phase is fully determined by this second component, as it will only contain the labels corresponding to disjuncts that are selected in the  $[\oplus]$  rules. Technically, this set of labels merely needs to be large enough to disambiguate all the disjunctive choices made in the positive phase.

**Definition 12.** A certificate for an LLKF sequent  $\sigma$  is of the form  $\langle \tau, \mathcal{D} \rangle$  where  $\tau$  is a neutral simply labelled L3KF sequent with  $\text{unl}(\tau) = \sigma$  and  $\mathcal{D}$  is a decision tree. A full certificate is a certificate with a full decision tree.

To consume (*i.e.*, check) a certificate, we execute the following algorithm.

**Definition 13 (checking proof certificates).** The following algorithm decides if a given proof certificate  $\langle \tau, \mathcal{D} \rangle$  is valid or not. We proceed by induction on the structure of  $\mathcal{D}$ .

1. If the root node of  $\mathcal{D}$  contains a contraction bound  $\mathcal{B}$ , then we enumerate all L3KF( $\mathcal{B}$ ) derivation of  $\tau$ , succeeding if any of them is a proof and failing otherwise.
2. If the root node of  $\mathcal{D}$  contains  $\langle \alpha, \Lambda \rangle$  and has children  $\mathcal{D}_1, \dots, \mathcal{D}_n$ , then we find  $\langle \alpha : P \rangle$  in  $\tau$  (failing if it doesn't exist) and perform the corresponding decision rule ( $[lf]$  or  $[uf]$ ). In the subsequent positive phase, for all disjunctive choices we select that disjunct whose immediate reactive subformulas have atomic labels found amongst  $\Lambda$  (failing if both disjuncts meet this criterion or if neither does). If this phase results in  $n$  neutral sequents  $\tau_1, \dots, \tau_n$ , we then check each certificate  $\langle \tau_i, \mathcal{D}_i \rangle$  for  $i \in 1..n$ .

In order for this checking procedure to avoid unnecessary work, the sub-trees in case 2 must line up precisely with the sub-derivations. This requires that the premises of an inference rule be produced in a predictable order, which in turn requires determinizing the order of application of the rules in the negative phase. This is easily done by treating the active context  $\Omega$  as a list instead of

as a multi-set, with the principal formula then always at the head of the list. This is precisely Andreoli’s original proposal for the negative (or asynchronous) phase in focusing [1]. Note that this is purely a matter of performance. The checking algorithm can backtrack over all the ways to match up sub-trees to neutral premises (there are only a finite number of permutations). The available indexes in the neutral premises and in the sub-trees can also give hints as to the right match-up.

**Theorem 3 (soundness of checking).** *If the certificate  $\langle \tau, \mathcal{D} \rangle$  is accepted by the algorithm of Defn. 13, then  $\text{unl}(\tau)$  is a provable LLKF sequent.*

*Proof.* Immediate from Thm. 2 (1). □

**Theorem 4 (completeness of certification).** *If  $\sigma$  is provable in LLKF, then there is a valid proof certificate  $\langle \tau, \mathcal{D} \rangle$  for which  $\text{unl}(\tau) = \sigma$ .*

*Proof (Sketch).* By Thm. 2 (2), there is an L3KF derivation of a  $\tau$  for some  $\tau$  with  $\text{unl}(\tau) = \sigma$ . The full decision tree from this L3KF derivation gives a suitable certificate. □

Observe that checking a full certificate involves no non-deterministic choices at all. There is, in fact, an order of determinacy among proof certificates, stated below as a theorem. Its proof is omitted here because it requires a fairly unilluminating sequence of technical lemmas.

**Proposition 1 (determinacy).** *Given two certificates  $\xi_1 = \langle \tau, \mathcal{D}_1 \rangle$  and  $\xi_2 = \langle \tau, \mathcal{D}_2 \rangle$ , say that  $\xi_1$  is more deterministic than  $\xi_2$  if  $\mathcal{D}_2$  with all contraction bound nodes removed is a prefix of  $\mathcal{D}_1$  likewise. Then, checking  $\xi_1$  involves fewer non-deterministic choices than checking  $\xi_2$ .* □

## 5 Concluding Remarks

We have given a way of systematically building proof certificates with a variable level of detail from focused sequent proofs. The main technical device is labelling of particular subformulas, which is both used to extract pre-computed information about contractions and to obtain a skeletal form of the proof as a *decision tree*. We have intentionally limited ourselves to bounded contraction as the mechanism for eliding detail from the proof certificate. There are obviously other—even simpler—means of eliding detail: for instance, instead of bounding contractions, we can just bound the overall depth of the sub-proof. These alternative mechanisms are readily compatible with the proposed design.

For intuitionistic or classical logic, the situation is generally simpler because of the absence of resource non-determinism. (Their propositional fragments are generally decidable anyway.) For the first-order case, the proof certificates would additionally have to depend on first-order unification in the consumer; alternatively, the decision tree nodes would have to record the existential witnesses.

As rightly pointed out by the anonymous referees, the claim in this paper of building compact proof certificates will ultimately have to be validated empirically. To this end, we are in the process of adapting the LI family of automated proof-producing linear logic provers [4] to function as “proof-elaborators” that will convert a certificate into a full proof, in essence implementing the algorithm of Defn. 13.

**Acknowledgement.** we thank Nicolas Guenot and Lutz Straßburger for many useful discussions on the nature of contraction, and the anonymous referees for their insightful comments.

## References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. *J. of Logic and Computation* 2(3), 297–347 (1992)
2. Boespflug, M.: Conception d’un noyau de vérification de preuves pour le  $\lambda II$ -calcul modulo. PhD thesis, Ecole Polytechnique (2011)
3. Brock-Nannestad, T., Schürmann, C.: Focused Natural Deduction. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 157–171. Springer, Heidelberg (2010)
4. Chaudhuri, K.: The Focused Inverse Method for Linear Logic. PhD thesis, Carnegie Mellon University, Technical report CMU-CS-06-162 (December 2006)
5. Chaudhuri, K.: Focusing Strategies in the Sequent Calculus of Synthetic Connectives. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 467–481. Springer, Heidelberg (2008)
6. Chaudhuri, K.: Magically Constraining the Inverse Method Using Dynamic Polarity Assignment. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 202–216. Springer, Heidelberg (2010)
7. Chaudhuri, K., Guenot, N., Straßburger, L.: The Focused Calculus of Structures. In: Computer Science Logic: 20th Annual Conference of the EACSL. Leibniz International Proceedings in Informatics (LIPIcs), pp. 159–173. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (September 2011)
8. Chaudhuri, K., Pfenning, F., Price, G.: A logical characterization of forward and backward chaining in the inverse method. *J. of Automated Reasoning* 40(2-3), 133–177 (2008)
9. Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic. *J. of Symbolic Logic* 57(3), 795–807 (1992)
10. Hodas, J., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. *Information and Computation* 110(2), 327–365 (1994)
11. Hodas, J., Watkins, K., Tamura, N., Kang, K.-S.: Efficient implementation of a linear logic programming language. In: Jaffar, J. (ed.) Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming, pp. 145–159 (1998)
12. Laurent, O.: Etude de la polarisation en logique. PhD thesis, Université Aix-Marseille II (March 2002)
13. Laurent, O.: A proof of the focalization property of linear logic (May 2004) (unpublished note)

14. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science* 410(46), 4747–4768 (2009)
15. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. *Annals Pure Applied Logic* 56, 239–311 (1992)
16. Miller, D.: A Proposal for Broad Spectrum Proof Certificates. In: Jouannaud, J.-P., Shao, Z. (eds.) *CPP 2011*. LNCS, vol. 7086, pp. 54–69. Springer, Heidelberg (2011)
17. Miller, D., Saurin, A.: From Proofs to Focused Proofs: A Modular Proof of Focalization in Linear Logic. In: Duparc, J., Henzinger, T.A. (eds.) *CSL 2007*. LNCS, vol. 4646, pp. 405–419. Springer, Heidelberg (2007)
18. Necula, G.C.: Proof-carrying code. In: *Conference Record of the 24th Symposium on Principles of Programming Languages 1997*, Paris, France, pp. 106–119. ACM Press (1997)
19. Straßburger, L.: *Linear Logic and Noncommutativity in the Calculus of Structures*. PhD thesis, Technische Universität Dresden (2003)



# Constructive Completeness for Modal Logic with Transitive Closure

Christian Doczkal and Gert Smolka

Saarland University, Saarbrücken, Germany  
{doczkal, smolka}@ps.uni-saarland.de

**Abstract.** Classical modal logic with transitive closure appears as a subsystem of logics used for program verification. The logic can be axiomatized with a Hilbert system. In this paper we develop a constructive completeness proof for the axiomatization using Coq with Ssreflect. The proof is based on a novel analytic Gentzen system, which yields a certifying decision procedure that for a formula constructs either a derivation or a finite countermodel. Completeness of the axiomatization then follows by translating Gentzen derivations to Hilbert derivations. The main difficulty throughout the development is the treatment of transitive closure.

**Keywords:** modal logic, completeness, decision procedures, constructive proofs, Hilbert Systems, Gentzen systems, Coq, Ssreflect.

## 1 Introduction

We are interested in a constructive and formal metatheory of decidable logics developed for program verification. In this paper we consider a logic  $K^+$ , which appears as a subsystem of PDL [8] and CTL [6].  $K^+$  extends the basic modal logic  $K$  with a modality for the transitive closure of the step relation. Our constructive account of  $K^+$  is based on a Hilbert system and a class of models. Our main result is a constructive proof that the Hilbert system is complete for our class of models. The completeness proof comes in the form of a certifying decision procedure<sup>1</sup> that for a formula constructs either a derivation in the Hilbert system or a finite countermodel. This establishes the completeness of the Hilbert system and the small model property of the logic. The main difficulty throughout the development is the treatment of transitive closure. The presence of the transitive closure modality for instance shows in the non-compactness of  $K^+$ .

We obtain our Hilbert system for  $K^+$  from a Hilbert system for PDL [15]. Proving the completeness of the Hilbert system constructively turned out to be a challenge. The completeness proofs in the literature [15] are based on maximal consistent sets and are thus nonconstructive. The notable exception is a paper [1] by Ben-Ari, Pnueli, and Manna, where the completeness of a Hilbert system for

---

<sup>1</sup> When we say decision procedure in this paper we do not claim that execution is feasible in practice.

UB (a logic subsuming  $K^+$ ) is shown by extending a tableau-based decision procedure such that it yields a Hilbert refutation in case it fails to construct a model.

We refine the approach of Ben-Ari et al. [1] by replacing the tableau-based system with an analytic Gentzen system. In contrast to the tableau system, which constructs models, the Gentzen system constructs derivations, which can be translated to Hilbert derivations. Less directly, the Gentzen system also constructs models. The states of the models are obtained from the underivable sequents containing only subformulas of the input formula. Thus the Gentzen system gives us a certifying decision procedure that for an input formula  $s$  returns either a finite model of  $\neg s$  or a derivation certifying that  $s$  is true in all models.

For propositional logic, the correspondence between tableau systems and Gentzen systems is immediate and well-known [17,20]. For modal logic, the situation is less clear. Fitting gives a tableaux system for S4 [9] and a corresponding Gentzen system [10]. This system can be easily adapted to  $K$ , and the resulting system serves as the basis of our Gentzen system for  $K^+$ . Finding the missing rule for  $K^+$  took some effort. Existing tableau systems for  $K^+$  and related logics use local conditions to expand the tableau but also check global conditions like reachability on the constructed graph. For a corresponding Gentzen system these conditions must be reformulated as inference rules deriving valid sequents from valid sequents. For  $K^+$  this leads to a rule we call compound rule. In contrast to the other rules, which are based on local properties, the compound rule in one step analyzes a strongly connected component of the search space for a model.

Once we have the Gentzen system for  $K^+$ , we translate Gentzen derivations into Hilbert derivations. To do so, we give for each Gentzen rule a function that for Hilbert derivations of the premises yields a Hilbert derivation of the conclusion. For all rules but the compound rule this is straightforward. The compound rule is the only rule dealing with the transitive closure modality. We handle the compound rule with an application of the induction-like Segerberg axiom of the Hilbert system to an invariant accounting for the strongly connected component licensing the application of the compound rule. Ben-Ari et al. [1] use the Segerberg axiom in a similar way but their invariant did not work for us.

Our development [4] is carried out in Coq [19] with the Ssreflect [14] extension. We profit much from Ssreflect since our development requires computational finite types for subformulas, sequents, and sets of sequents.

We think that certifying decision procedures for logics used in program verification deserve more attention. The usual tableau-based decision procedures for such logics (e.g., [1,16]) construct finite models for satisfiable formulas but do not construct certificates for unsatisfiable formulas. For  $K^+$ , we remedy this situation with a Gentzen system that constructs derivations for unsatisfiable formulas and models for satisfiable formulas.

In a previous paper [5] we give a constructive and formal proof of the decidability of an extension of  $K^+$ . There we rely on a pruning-based decision method and make no attempt to generate Hilbert proofs.

The paper is organized as follows. We first define the syntax, the models, and the Hilbert system for  $K^+$ . We then say a few things about finite types and finite sets in `Ssreflect`, which provide essential infrastructure for our development. Next we discuss how we formalize analytic Gentzen systems in `Coq` using `Ssreflect`. We then define a class of syntactic models we call *demos*. *Demos* represent the models produced by our Gentzen systems. Next we introduce propositional retracts, which we need for the formulation of the compound rule. We then define the Gentzen system for  $K^+$  and prove that it yields a *demo* for every undervivable sequent. Finally, we show how derivations in this system are translated to derivations in the Hilbert system.

## 2 Problem Statement

We assume a countable alphabet  $\mathcal{P}$  of atomic propositions  $p$  and consider the formulas

$$s, t ::= \perp \mid p \mid s \rightarrow t \mid \Box s \mid \Box^+ s$$

To increase readability, we introduce a number of defined logical operations.

$$\neg s := s \rightarrow \perp \quad s \wedge t := \neg(s \rightarrow \neg t) \quad s \vee t := \neg s \rightarrow t \quad \Box^* s := s \wedge \Box^+ s$$

Formulas are interpreted over transition systems consisting of a set of states  $|\mathcal{M}|$ , a transition relation  $\rightarrow_{\mathcal{M}} \subseteq |\mathcal{M}| \times |\mathcal{M}|$ , and a labeling  $\Lambda_{\mathcal{M}} : |\mathcal{M}| \rightarrow 2^{\mathcal{P}}$ . The *satisfaction relation*  $\mathcal{M}, w \models s$  between transition systems, their states, and formulas is defined as follows:

$$\begin{aligned} \mathcal{M}, w &\not\models \perp \\ \mathcal{M}, w &\models p \iff p \in \Lambda_{\mathcal{M}}(w) \\ \mathcal{M}, w &\models s \rightarrow t \iff \mathcal{M}, w \models s \text{ implies } \mathcal{M}, w \models t \\ \mathcal{M}, w &\models \Box s \iff \mathcal{M}, v \models s \text{ for all } v \text{ such that } w \rightarrow_{\mathcal{M}} v \\ \mathcal{M}, w &\models \Box^+ s \iff \mathcal{M}, v \models s \text{ for all } v \text{ such that } w \rightarrow_{\mathcal{M}}^+ v \end{aligned}$$

Here,  $\rightarrow_{\mathcal{M}}^+$  is the transitive closure of the transition relation. In `Coq`, we represent transition systems as a record type:

```
Record ts : Type := TS {
  state :> Type ;
  trans : state -> state -> Prop ;
  label : state -> aprop -> Prop }.
```

We define the satisfaction relation as a function into `Prop`:

```
satisfies : forall (T : ts), T -> form -> Prop
```

Since we consider classical modal logic, we consider as *models* those transition systems, for which the satisfaction relation is stable under double negation.

```
Definition stable (X Y : Type) (R : X -> Y -> Prop) :=
```

```
forall x y, ~ ~ R x y -> R x y.
```

```
Record model := Model { ts_of :> ts ; modelP : stable (satisfies ts_of) }.
```

$$\begin{array}{ll}
s \rightarrow t \rightarrow s & \text{(K)} \\
(s \rightarrow t \rightarrow u) \rightarrow (s \rightarrow t) \rightarrow (s \rightarrow u) & \text{(S)} \\
\neg \neg s \rightarrow s & \text{(DN)} \\
\Box(s \rightarrow t) \rightarrow \Box s \rightarrow \Box t & \text{(N)} \\
\Box^+(s \rightarrow t) \rightarrow \Box^+ s \rightarrow \Box^+ t & \text{(N+)} \\
\Box^+ s \rightarrow \Box s & \text{(T1)} \\
\Box^+ s \rightarrow \Box \Box^+ s & \text{(T2)} \\
\Box s \rightarrow \Box \Box^+ s \rightarrow \Box^+ s & \text{(T3)} \\
\Box s \rightarrow \Box^+(s \rightarrow \Box s) \rightarrow \Box^+ s & \text{(Segeberg)} \\
\\
\frac{s \rightarrow t \quad s}{t} \text{MP} & \frac{s}{\Box s} \text{NEC} & \frac{s}{\Box^+ s} \text{NEC}^+
\end{array}$$

**Fig. 1.** Hilbert System for Modal Logic with Transitive Closure

A formula is *satisfiable* if it has a model, i.e., it holds at some state of some model. A formula  $s$  is *valid* if it holds at every state of every model. The Hilbert system for which we want to show completeness is shown in Figure 1. We write  $\vdash s$  if  $s$  is provable in the Hilbert system. We represent the Hilbert system in Coq as an inductive predicate:

```

Inductive prv : form -> Prop :=
| r_mp s t      : prv (s ----> t) -> prv s -> prv t
| ax_k s t      : prv (s ----> t ----> s)
...

```

We can immediately show soundness.

**Lemma 2.1 (Soundness).** *If  $s$  is provable, then  $s$  is valid.*

*Proof.* Induction on the derivation of  $\vdash s$ , using stability of the satisfaction relation to show the case for (DN).

Our main result is the following.

**Theorem 2.2 (Certified Decidability).** *For every formula  $s$ , we can either construct a finite model of  $\neg s$  or a proof of  $s$ .*

**Corollary 2.3 (Completeness).** *If  $s$  is valid, then  $s$  is provable.*

For the rest of this paper, we will mostly use mathematical notation to convey the ideas of the completeness proof. We present Coq code to show design choices and when the formal proof differs from the mathematical presentation. The reader is invited to browse the `coqdoc` proof outline and the source files [4].

### 3 Finite Types and Finite Sets in Ssreflect

Our formal proofs rely heavily on the Ssreflect extension to Coq, so we briefly describe the most important features we use. For technical details refer to [12,13].

<code>   </code>	<code>&amp;&amp;</code>	Boolean disjunction and conjunction
<code>x \in xs</code>		Generic membership operation for lists, sets, etc.
<code>seq_sub xs</code>		The finite type whose elements are the members of the list <code>xs</code>
<code>{set X}</code>		The finite type of sets over the finite type <code>X</code> .
<code>: </code>	<code>:&amp;</code>	Union and intersection
<code>[set x:X   A]</code>		The set $\{x \in X \mid A\}$ as an element of the type <code>{set X}</code> .

Fig. 2. Ssreflect’s Syntax for Finite Types and Finite Sets

In Ssreflect, a counted type is a type with a boolean equality test and a choice operator for boolean predicates. A finite type is a counted type together with a finite list enumerating its elements. Finite types can be constructed from finite lists and finiteness is preserved by many type constructors. In particular, finite types are closed under cartesian products and taking sets. Finite types come with boolean quantifiers `[forall x, p x]` and `[exists x, p x]` taking boolean predicates and returning booleans. The Coq syntax for the remaining operations we use is given in Figure 2.

## 4 Analytic Gentzen Systems in Coq

For a constructive proof of Theorem 2.2, we need a decision procedure which for a given formula either constructs a countermodel or a Hilbert proof. We will use an analytic Gentzen system for this purpose. Before we develop the Gentzen system for  $K^+$ , we first show how we represent analytic Gentzen systems in Coq. As an example we will use an analytic Gentzen for basic modal logic  $K$  which is adapted from Fitting’s tableau system for  $S4$  [9]. This Gentzen system for  $K$  will also serve as the starting point for the development of our Gentzen System for  $K^+$ .

We represent sequents as finite sets of *signed formulas* [17] we call *clauses*. For instance, the sequent  $p, q \Rightarrow u, v$  is represented as the clause  $\{p^+, q^+, u^-, v^-\}$ . The letter  $C$  ranges over clauses. A state *satisfies* a signed formula  $s^\sigma$  if it satisfies  $[s^\sigma]$  where  $[s^+] = s$  and  $[s^-] = \neg s$ . A state satisfies a clause, if it satisfies all signed formulas it contains. Accordingly, the *associated formula* of a clause  $C$  is  $\bigwedge_{s^\sigma \in C} [s^\sigma]$ .

A sound Gentzen system is now a deduction system that derives unsatisfiable clauses from unsatisfiable clauses. This is in harmony with the conventional view that a sound Gentzen system derives valid sequents from valid sequents since validity of the formula  $p \wedge q \rightarrow u \vee v$  associated with the sequent  $p, q \Rightarrow u, v$  is equivalent to unsatisfiability of the formula we associate to the clause  $\{p^+, q^+, u^-, v^-\}$

Figure 3 shows our Gentzen system for basic modal logic  $K$ . The notation  $C; s^\sigma$  is to be read as  $C \cup \{s^\sigma\}$ . The notation  $\mathcal{RC}$  denotes the set  $\{s^+ \mid \Box s^+ \in C\}$ , which we call the *request* of  $C$ . The system is *analytic* in the sense that if a clause  $C$  is derivable, it has a derivation employing only signed subformulas of the

$$\begin{array}{c}
 \frac{}{C; s^+; s^-} \text{Ax} \qquad \frac{}{C; \perp^+} \perp \\
 \\
 \frac{C; s^+; t^-}{C; s \rightarrow t^-} \rightarrow^- \qquad \frac{C; s^- \quad C; t^+}{C; s \rightarrow t^+} \rightarrow^+ \qquad \frac{RC; s^-}{C; \Box s^-} \text{JUMP}
 \end{array}$$

**Fig. 3.** Analytic Gentzen System for K

formulas in  $C$ . For analytic Gentzen systems derivability of clauses is decidable provided that rule instantiation is decidable.

We want to use analytic Gentzen systems as certifying decision procedures. Hence, we fix an “input” formula  $s_0$  and parameterize our definitions by this formula. This allows us to only consider the finitely many signed subformulas of  $s_0$ , which in turn allows us to leverage `Ssreflect`’s finite types and sets.

Writing `sub s0` for the list of subformulas of  $s_0$ , we represent the signed formulas as the finite type `F` and clauses as sets over `F`.

**Definition** `F := (bool * seq_sub (sub s0)) %type.`

**Definition** `clause := {set F}.`

This allows most properties of clauses and sets of clauses to be expressed as boolean predicates and reasoned about classically. The rules of a Gentzen system will be represented as a boolean predicate

```
rule : {set clause} -> clause -> bool
```

Thus, the type of the rule predicate ensures that the system is analytic. Using a boolean predicate to represent rules also captures our intuition that rule instantiation should be decidable.

The set of derivable clauses is the least fixpoint of one-step derivability:

```
Definition onestep_derivable_from (S : {set clause}) :=
  [set C | [exists D : {set clause}, (D \subset S) && rule D C]].
```

One-step derivability is monotone so the least fixpoint exists and can be computed by applying the `onestep_derivable_from` function  $n$  times to the empty set where  $n$  is the size of the type `clause`. Hence derivability is decidable for any boolean rule predicate.

## 5 Demos

We now define a class of syntactic models we call `demos` [16]. The states of `demos` will be clauses and the definition will be such that every `demo` satisfies all the clauses it contains. Further, we will design our Gentzen system for  $K^+$  such that it is complete for `demos`, i.e., the underivable clauses contain a `demo` satisfying all underivable clauses.

A clause  $H$  is called a *Hintikka clause* if it satisfies the following conditions:

- H1.  $\perp^+ \notin H$
- H2. If  $p^- \in H$ , then  $p^+ \notin H$
- H3. If  $(s \rightarrow t)^+ \in H$ , then  $s^- \in H$  or  $t^+ \in H$
- H4. If  $(s \rightarrow t)^- \in H$ , then  $s^+ \in H$  and  $t^- \in H$

To express the Hintikka property as a boolean predicate we have to take care of the fact that the Coq representation of our signed formulas consist of three parts: a formula, a proof that this formula is a subformula of  $s_0$ , and a boolean sign. With  $\text{sc} : s \ \backslash\text{in} \ \text{sub} \ s_0$  we write the signed formula  $s^-$  as  $[\text{F } s; \text{sc}; \text{false}]$ . We define projections on the membership proofs in  $\text{sub} \ s_0$ :

**Lemma**  $\text{pI1 } s \ t \ (\text{sc} : s \ \text{---} \rightarrow t \ \backslash\text{in} \ \text{sub} \ s_0) : s \ \backslash\text{in} \ \text{sub} \ s_0$ .

With this in place, we can express the Hintikka property as follows:

**Definition**  $\text{Hcond } (t : F) (H : \{\text{set } F\}) :=$   
 $\text{match } t \ \text{with}$   
 $\quad | [\text{F } s \ \text{---} \rightarrow t; \text{sc}; \text{true}] \ \Rightarrow$   
 $\quad \quad ([\text{F } s; \text{pI1 } \text{sc}; \text{false}] \ \backslash\text{in} \ H) \ || \ ([\text{F } t; \text{pIr } \text{sc}; \text{true}] \ \backslash\text{in} \ H)$   
 $\quad | \dots$   
 $\text{end.}$

**Definition**  $\text{hintikka } (H : \{\text{set } F\}) : \text{bool} := [\text{forall } t \ \text{in} \ H, \text{Hcond } t \ H]$ .

We now come to the definition of demos. The states of demos are Hintikka clauses. For the transition relation we extend the notion of *request* to  $K^+$ .

$$\mathcal{RC} := \{s^+ \mid \Box s^+ \in C\} \cup \{s^+ \mid \Box^+ s^+ \in C\} \cup \{\Box^+ s^+ \mid \Box^+ s^+ \in C\}$$

For every set  $\mathcal{S}$  of clauses, we define a *transition relation*  $\rightarrow_{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$  as follows:  $C \rightarrow_{\mathcal{S}} D$  iff  $\mathcal{RC} \subseteq D$  and  $\{C, D\} \subseteq \mathcal{S}$ . We write  $\rightarrow_{\mathcal{S}}^+$  for the transitive closure of  $\rightarrow_{\mathcal{S}}$ . A set  $\mathcal{D}$  of Hintikka clauses is a *demo* if every clause  $C \in \mathcal{D}$  satisfies the following conditions:

- D1. If  $\Box t^- \in C$ , then there is a clause  $D \in \mathcal{D}$  such that  $C \rightarrow_{\mathcal{D}} D$  and  $t^- \in D$ .
- D2. If  $\Box^+ t^- \in C$ , then there is a clause  $D \in \mathcal{D}$  such that  $C \rightarrow_{\mathcal{D}}^+ D$  and  $t^- \in D$ .

A *demo for a clause*  $C$  is a demo that contains a clause that extends  $C$ . Let  $\mathcal{D}$  be a demo. The *model associated with*  $\mathcal{D}$  takes as states the clauses in  $\mathcal{D}$  and as transition relation the relation  $\rightarrow_{\mathcal{D}}$ . Moreover, a state  $C$  of the associated model is labeled with atomic proposition  $p$  if and only if  $p \in C$ .

**Lemma 5.1.** *Let  $\mathcal{D}$  be a demo,  $\mathcal{M}$  its associated model, and  $s$  a formula. If  $s^\sigma \in C \in \mathcal{D}$ , then  $\mathcal{M}, C \models [s^\sigma]$ .*

*Proof.* Induction on  $s$ . □

Note that, in contrast to the demo condition for  $\Box t^-$ , the demo condition for  $\Box^+ t^-$  is non-local in the sense that it may take an arbitrary number of transitions to reach the clause containing  $t^-$ . We call a formula of the form  $\Box^+ t^-$  an *eventuality* and say that a clause  $D$  satisfying (D2) *fulfills* the eventuality. Coming up with a Gentzen system whose underivable Hintikka clauses satisfy (D2) requires some work.

## 6 Propositional Retracts

We now begin the development of our Gentzen system for  $K^+$ . To ease our notation, we will from now on write just  $s$  for a positively signed formula  $s^+$ .

We will design our system such that the set of underivable clauses will contain a demo for every underivable clause. Since demos only contain Hintikka clauses, we need to ensure that for every underivable clause there is an underivable Hintikka extension, i.e, an underivable Hintikka clause containing  $C$ .

We call a set of clauses  $\mathcal{D}$  a *retract* of  $C$ , written  $\mathcal{D} \triangleright C$ , if  $C$  is derivable from  $\mathcal{D}$  using propositional reasoning. More precisely, a retract of  $C$  is the frontier of a backwards derivation starting at  $C$  using the propositional rules from Figure 3. See Figure 4 for the precise definition of this notion. A retract  $\mathcal{D}$  of  $C$  is called a *Hintikka retract* if every  $H \in \mathcal{D}$  is a Hintikka extension of  $C$ .

**Lemma 6.1.** *For every clause one can compute a Hintikka retract.*

*Proof.* Let  $C$  be a clause. We prove this by induction on the number of signed formulas not in  $C$ . If  $C$  is a Hintikka clause, then  $\{C\}$  is a Hintikka retract of  $C$ . If  $C$  contains a formula both with positive and negative sign or  $\perp^+$ , we pick the empty retract. Otherwise, there is some implication  $s \rightarrow t^\sigma$  whose Hintikka condition is not satisfied. We consider the case where  $\sigma = -$ ; the other case is similar. We have  $C \subseteq C; s; t^-$  so by induction hypothesis we can compute a Hintikka retract  $H$  for  $C; s; t^-$  which is also a Hintikka retract for  $C$ .  $\square$

On the Coq side, this amounts to showing.

**Lemma saturation C : { H | hretract H C }**

Here,  $\{ H \mid \text{retract } H \mathcal{D} \}$  is the type of dependent pairs of sets of clauses  $H$  and proofs that these are Hintikka retracts of  $C$ . We define a function which computes a Hintikka retract for every clause, by projecting out the first component of that pair:

**Definition dret C := proj1\_sig (saturation C)**

We refer to the Hintikka retract computed by **dret** as the *default retract* of  $C$ . We now have the first rule of our Gentzen system for  $K^+$ , the *retract rule*:

$$\frac{C_1 \dots C_n}{C} \{C_1, \dots, C_n\} \text{ is the default Hintikka retract for } C$$

$$\frac{}{\{C\} \triangleright C} \qquad \frac{}{\emptyset \triangleright C} \perp \in C \text{ or } \{s^+, s^-\} \subseteq C$$

$$\frac{D_1 \triangleright C; s^- \quad D_2 \triangleright C; t}{D_1 \cup D_2 \triangleright C; s \rightarrow t^+} \qquad \frac{D \triangleright C; s; t^-}{D \triangleright C; s \rightarrow t^-}$$

**Fig. 4.** Retracts



Note that the use of the default retract in the definition of the retract rule essentially fixes a single strategy in which the propositional rules can be applied. This allows us to express the retract rule as a boolean predicate without the tedium of expressing the retract relation as a boolean predicate. Mathematically, any Hintikka retract would suffice since we do not make use of any special properties of the default retract.

**Lemma 6.2 (Extension).** *If  $C$  is underivable, it has an underivable Hintikka extension.*

Note that all clauses that are derivable using propositional reasoning have an empty default retract. Hence, the retract rule allows us to handle propositional reasoning in a single step and completely separately from modal reasoning.

## 7 The Gentzen System for $K^+$

The Gentzen system consisting of the retract rule and the jump rule from Figure 3 is already complete for formulas not involving  $\Box^+$ .

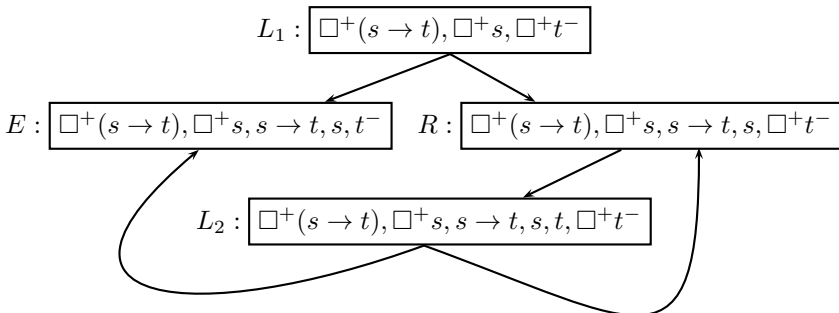
**Lemma 7.1.** *The underivable Hintikka clauses satisfy condition (D1).*

Hence, all we need is a rule that establishes (D2). A first candidate for a rule for the  $\Box^+$  modality might be the following *split-jump rule*:

$$\frac{\mathcal{RC}; s^- \quad \mathcal{RC}; \Box^+ s^-}{C; \Box^+ s^-}$$

The rule is motivated by the two ways in which we can satisfy the eventuality  $\Box^+ s^-$ : we either satisfy  $s^-$  at the next state (left branch of the rule) or we delay fulfilling the eventuality (right branch). However, the resulting Gentzen system would be incomplete as witnessed by the following example.

*Example 7.2.* The clause  $\{\Box^+(s \rightarrow t), \Box^+ s, \Box^+ t^-\}$  corresponding to the axiom  $(N^+)$  is an example of an unsatisfiable but underivable clause. Alternating between applying the split-jump and the retract rule yields a looping derivation that can be visualized as a graph:



Here,  $\{L_2\}$  is the unique Hintikka retract of  $R$ . The clause  $E$  can be proved by the retract rule, but the derivation “loops” around the clause  $L_2$ .

To obtain a complete deduction system, we need a stronger rule for eventualities. As we have seen in the previous example, applying the retract and the jump rule can lead to looping derivations on unsatisfiable clauses. Thus, our rule for eventualities is a generalization of the split-jump rule that allows for certain looping derivations.

A *compound* is a triple  $(s, \mathcal{L}, \mathcal{E})$  consisting of a formula  $s$  and two sets  $\mathcal{L}$  and  $\mathcal{E}$  of clauses such that every clause  $C \in \mathcal{L}$  satisfies the following conditions:

1.  $\Box^+s^- \in C$
2.  $\mathcal{R}C; s^- \in \mathcal{E}$
3.  $\mathcal{L} \cup \mathcal{E}$  contains all clauses of the default retract of  $\mathcal{R}C; \Box^+s^-$ .

If  $(s, \mathcal{L}, \mathcal{E})$  is a compound, we call  $\mathcal{L}$  the *loop* and the clauses in  $\mathcal{E}$  *exit clauses*. We now formulate the remaining rule and call it *compound rule*:

$$\frac{C_1 \cdots C_n}{C} \quad (s, \mathcal{L}, \{C_1, \dots, C_n\}) \text{ is a compound and } C \in \mathcal{L}$$

Starting from a clause  $C$  containing  $\Box^+s^-$ , instances of the compound rule can be generated as follows: We set  $\mathcal{L} := \{C\}$  and  $\mathcal{E} := \emptyset$  and start by applying the split-jump rule to  $C$ . The left premise and those clauses in the default retract of  $\mathcal{R}C; \Box^+s^-$  that we want to derive by other means are added to  $\mathcal{E}$ . The remaining clauses that are not yet in  $\mathcal{L}$  are added to  $\mathcal{L}$ , and we continue by applying the split-jump rule to these clauses. This process must terminate with a compound  $(s, \mathcal{L}, \mathcal{E})$  since there are only finitely many clauses that can be added to  $\mathcal{L}$ . The fact that one does not need to apply the split-jump rule to clauses that are already in  $\mathcal{L}$  allows for looping derivations. The compound rule can easily be expressed as a boolean predicate.

*Example 7.3.* We can derive the clause  $\{\Box^+(s \rightarrow t), \Box^+s, \Box^+t^-\}$  from Example 7.2 using the compound rule. Taking  $L_1, L_2$ , and  $E$  as in Example 7.2, the triple  $(t, \{L_1, L_2\}, \{E\})$  is a compound and  $E$  can be derived using the retract rule.

**Lemma 7.4.** *The underivable Hintikka clauses satisfy condition (D2).*

*Proof.* Let  $\mathcal{U}$  be the set of underivable Hintikka clauses and  $\Box^+s^- \in C \in \mathcal{U}$ . We define

$$\mathcal{L} := \{C' \in \mathcal{U} \mid \Box^+s^- \in C \text{ and } C \rightarrow_{\mathcal{U}}^* C'\}$$

Let  $\mathcal{D}$  be the set of derivable clauses. Since  $C \in \mathcal{L}$  and  $C$  is not derivable, we know that  $(s, \mathcal{L}, \mathcal{D})$  is not a compound. However, compound conditions (1) and (3) hold. Hence there is a clause  $D \in \mathcal{L}$  such that  $\mathcal{R}D; s^- \notin \mathcal{D}$ . Thus  $\mathcal{R}D; s^-$  is not derivable and therefore has a Hintikka extension  $D' \in \mathcal{U}$  (Lemma 6.2). Thus  $C \rightarrow_{\mathcal{U}}^* D \rightarrow_{\mathcal{U}} D'$  and  $t^- \in D'$ .  $\square$

We now have a complete Gentzen system for  $K^+$ .

**Theorem 7.5.** *The set of underivable Hintikka clauses is a demo for every underivable clause. Thus every underivable clause is satisfiable.*

*Proof.* Follows with Lemma 7.1, Lemma 7.4, and Lemma 5.1 □

## 8 Translating Gentzen Derivations to Hilbert Proofs

So far we did not consider soundness of the Gentzen system. However, soundness of the Gentzen system will follow as a by-product of a translation of Gentzen derivations to Hilbert proofs, which we need anyway to show completeness of the Hilbert system.

We associate with every signed formula and every clause a formula as defined in Section 4. If a clause appears in the place of a formula, the clause is to be understood as notation for its associated formula.

We aim for a translation theorem of the following form:

**Theorem 8.1.** *If  $C$  is derivable in the Gentzen system, then  $\vdash \neg C$ .*

We call a Hilbert proof of  $\neg C$  a (Hilbert) *refutation* of  $C$ . A constructive proof of the translation theorem can be seen as a translation from Gentzen derivations to Hilbert refutations. To prove this theorem by induction on the Gentzen derivation, we need a number of lemmas corresponding to the rules of the Gentzen system.

For the retract rule we have the following Lemma which we will also use in the translation of the compound rule.

**Lemma 8.2.** *If  $\mathcal{D} \triangleright C$ , then  $\vdash C \rightarrow \bigvee \mathcal{D}$ .*

*Proof.* Induction on the definition of retract. □

For the jump rule we have:

**Lemma 8.3.** *If  $\vdash \neg(\mathcal{R}C; s^-)$ , then  $\vdash \neg(C; \Box s^-)$ .*

*Proof.* We reason as follows:

- |  |  |
|--|--|
| 1. $\vdash \neg(\mathcal{R}C; s^-)$              | assumption                               |
| 2. $\vdash \mathcal{R}C \rightarrow s$           | propositional reasoning                  |
| 3. $\vdash \Box \mathcal{R}C \rightarrow \Box s$ | NEC, N                                   |
| 4. $\vdash C \rightarrow \Box s$                 | $\vdash C \rightarrow \Box \mathcal{R}C$ |
| 5. $\vdash \neg(C; \Box s^-)$                    | propositional reasoning                  |

□

Note that the Gentzen system consisting only of the retract rule and the jump rule corresponds very closely to the Gentzen system in 3 and is complete for formulas not involving  $\Box^+$ . So giving a constructive completeness proof for  $K$  is not difficult. The difficulty of giving a constructive completeness proof for  $K^+$  lies entirely in the treatment of transitive closure.

## 9 Generating Hilbert Proofs

Before we turn to the translation of the compound rule, we first note that to formalize this kind of translation argument, we need to develop some infrastructure for generating Hilbert proofs in Coq as finding Hilbert proofs in the bare Hilbert system can be a difficult task. Of course, Hilbert systems are well understood and there are many techniques to come up with Hilbert proofs. We merely mention two techniques that are easy to set up and help significantly in generating Hilbert proofs in Coq.

It is well known that the entailment relation (i.e.,  $\text{prv } (s \dashv\vdash t)$  in Coq) defines a preorder on formulas and that the logical operations have certain *monotonicity properties* with respect to this preorder. For example:

$$\frac{\vdash s' \rightarrow s \quad \vdash t \rightarrow t'}{\vdash (s \rightarrow t) \rightarrow (s' \rightarrow t')} \quad \frac{\vdash s \rightarrow s' \quad \vdash t \rightarrow t'}{\vdash (s \wedge t) \rightarrow (s' \wedge t')} \quad \frac{\vdash s \rightarrow s'}{\vdash \Box s \rightarrow \Box s'} \quad \dots$$

We make these monotonicity properties known to Coq’s extended (setoid) rewriting tactic [18]. This allows us to freely rewrite with the entailment relation to strengthen claims or to weaken assumptions, thereby contracting many mechanical reasoning steps into a single rewrite.

Another major hindrance, in particular to finding propositional proofs, is the lack of assumption management in the Hilbert system. However, we can simulate natural deduction style reasoning inside the Hilbert system using a few lemmas on big conjunctions.

In Coq, we realize big conjunctions and disjunctions over lists of formulas using Ssreflect’s canonical big operators [2]. We also need big conjunctions indexed by finite sets, which we represent by fixing an arbitrary enumeration of the elements. The most prominent use of this construction is for the associated formulas of clauses. For a big conjunction of the form  $\bigwedge_{x \in A} x$  we will just write  $\bigwedge A$  and likewise for disjunctions.

The lemmas we use to simulate natural deduction style reasoning are displayed as rules in Figure 5. Here  $xs$  ranges over lists of formulas and  $::$  is the cons operator. Note that there are no rules corresponding to NEC and NEC<sup>+</sup> since these rules would clearly be unsound in the presence of assumptions. This essentially restricts reasoning with assumptions to the propositional fragment, which is sufficient for our purposes.

Building on these rules, we define a set of tactics simulating the behavior of basic Coq tactics like `intros` and `apply` on the level of Hilbert proofs. For additional detail, we refer the reader to the theory files. Using setoid rewriting and these tactics the various modal logic lemmas that we need for our translation proof can be proved easily. Those modal logic lemmas to which we will refer explicitly can be found in Figure 6.

## 10 Translation Method for the Compound Rule

We now turn to the last missing piece in our formal completeness proof, the translation method for the compound rule.

$$\begin{array}{c}
 \frac{s \in xs}{\bigwedge xs \rightarrow s} \qquad \frac{\bigwedge s :: xs \rightarrow t}{\bigwedge xs \rightarrow s \rightarrow t} \qquad \frac{\bigwedge nil \rightarrow s}{s} \\
 \\
 \frac{\bigwedge xs \rightarrow s \rightarrow t \quad \bigwedge xs \rightarrow s}{\bigwedge xs \rightarrow t}
 \end{array}$$

**Fig. 5.** Assumption Lemmas

$$\begin{array}{ll}
 \vdash \Box s \vee \Box t \rightarrow \Box(s \vee t) & \text{(D2)} \\
 \vdash \Box^* s \rightarrow \Box \Box^* s & \text{(S1)} \\
 \vdash \Box^+ s \rightarrow \Box \Box^* s & \text{(S2)} \\
 \vdash C \rightarrow \Box \mathcal{R}C & \text{(R1)} \\
 \text{If } \vdash s \rightarrow t, \text{ then } \vdash \Box^+ s \rightarrow \Box^+ t & \text{(REG}^+\text{)}
 \end{array}$$

**Fig. 6.** Basic Modal Logic Lemmas

**Lemma 10.1.** *Let  $(s, \mathcal{L}, \mathcal{E})$  be a compound such that  $\vdash \neg D$  for all  $D \in \mathcal{E}$ . Then for every  $C \in \mathcal{L}$ , we have  $\vdash \neg C$ .*

*Proof.* Let  $C_0 \in \mathcal{L}$ . It suffices to show  $\vdash C_0 \rightarrow \Box^+ s$  since  $\Box^+ s^- \in C_0$  by the definition of compound. We define

$$I := \Box^* s \vee \bigvee_{C \in \mathcal{L}} \mathcal{R}C$$

and show the following properties of  $I$ :

- (i)  $\vdash C_0 \rightarrow \Box I$
- (ii)  $\vdash I \rightarrow s$
- (iii)  $\vdash I \rightarrow \Box I$

Once we have shown (i) to (iii), we can finish the proof as follows:

- 1.  $\vdash \Box^+(I \rightarrow \Box I)$  (iii), NEC<sup>+</sup>
- 2.  $\vdash \Box I \rightarrow \Box^+ I$  Segerberg
- 3.  $\vdash \Box I \rightarrow \Box^+ s$  (ii), REG<sup>+</sup>
- 4.  $\vdash C_0 \rightarrow \Box^+ s$  (i)

For (i - iii) we reason as follows:

(i) We have:

- 1.  $\vdash C_0 \rightarrow \Box \mathcal{R}C_0$  (R1)
- 2.  $\vdash C_0 \rightarrow \Box I$  monotonicity

(ii) It suffices to show  $\vdash \mathcal{R}C \rightarrow s$  for every  $C \in \mathcal{L}$ . For every such  $C$  we have:

1.  $\vdash \neg(\mathcal{R}C; s^-)$  Def. compound, assumption
2.  $\vdash \mathcal{R}C \rightarrow s$  propositional reasoning

(iii) We show that every disjunct of  $I$  implies  $\Box I$ . By (S1) it suffices to show  $\vdash \mathcal{R}C \rightarrow \Box I$  for every  $C \in \mathcal{L}$ . Let  $C \in \mathcal{L}$  and let  $\mathcal{D} \subseteq \mathcal{L} \cup \mathcal{E}$  be the default retract of  $\mathcal{R}C; \Box^+ s^-$ .

1.  $\vdash \mathcal{R}C; \Box^+ s^- \rightarrow \bigvee \mathcal{D}$  Lemma 8.2
2.  $\vdash \mathcal{R}C; \Box^+ s^- \rightarrow \bigvee (\mathcal{D} \cap \mathcal{L})$   $\vdash \neg D$  for all  $D \in \mathcal{E}$
3.  $\vdash \mathcal{R}C; \Box^+ s^- \rightarrow \bigvee_{L \in \mathcal{L}} L$  propositional reasoning
4.  $\vdash \mathcal{R}C; \Box^+ s^- \rightarrow \bigvee_{L \in \mathcal{L}} \Box RL$  (R1), monotonicity
5.  $\vdash \mathcal{R}C; \Box^+ s^- \rightarrow \Box \bigvee_{L \in \mathcal{L}} RL$  (D2)
6.  $\vdash \mathcal{R}C \rightarrow \Box^+ s \vee \bigvee_{L \in \mathcal{L}} RL$  propositional reasoning
7.  $\vdash \mathcal{R}C \rightarrow \Box(\Box^* s \vee \bigvee_{L \in \mathcal{L}} RL)$  (S2), (D2)
8.  $\vdash \mathcal{R}C \rightarrow \Box I$  Def. I

□

Now we can prove the translation theorem.

*Proof (of Theorem 8.1).* Let  $C$  be derivable. We prove the claim by induction on the derivation of  $C$ . The case for the compound rule follows immediately with Lemma 10.1. The cases for the jump rule and the retract rule follow with Lemma 8.3 and Lemma 8.2 respectively. □

*Proof (of Theorem 2.2).* Derivability in our Gentzen system is decidable, so we consider two cases:

*The clause  $\{s^-\}$  is derivable.* By Lemma 8.1, we have  $\vdash \neg\neg s$  and hence  $\vdash s$ .

*The clause  $\{s^-\}$  is underivable.* By Lemma 7.5, the set of underivable clauses over the subformulas of  $s$  yields a model for  $\{s^-\}$  and hence for  $\neg s$ . □

Note that in the proof above, the size of the countermodel can be bounded by  $2^{2^n}$  where  $n$  is the number of subformulas of  $s$ . Hence, we have also shown that  $K^+$  has the small model property.

Organizing the proof as we do, we obtain a development of rather moderate size. It consists of less than 1000 lines specific to our proof, about half of which are Hilbert infrastructure and a collection of basic modal logic lemmas. On top of this we need a couple of hundred lines of generic constructions like the fixpoint computation described in Section 4.

## 11 Related Work

Only after submitting the initial version of this paper, we became aware of the work of Brännler and Lange [3] presenting analytic sequent calculi for LTL and CTL. In their calculi one can focus on an eventuality formula and keep a history of the contexts in which a rule has been applied to this eventuality. If an eventuality occurs in a context that is already in the history, the sequent is provable.

Adapting Brännler and Lange’s rules to  $K^+$ , we obtain a system where one of the eventuality formulas in a clause can be annotated with a finite set of plain, i.e., annotation-free clauses. The rules for  $\Box^+$  then look as follows:

$$\frac{\mathcal{RC}; s^-}{C; \Box^+ s^-} \quad \frac{\mathcal{RC}; \Box^+_{\mathcal{RC}} s^-}{C; \Box^+ s^-} \quad \frac{\mathcal{RC}; s^- \quad \mathcal{RC}; \Box^+_H \mathcal{RC} s^-}{C; \Box^+_H s^-} \quad \frac{}{C; \Box^+_H \mathcal{RC} s^-}$$

An annotated eventuality  $\Box^+_H s^-$  is satisfied at a state  $w$  if there is a path from  $w$  to a state satisfying  $s^-$  that has at least one transition and after the first transition no state on the path satisfies a clause in  $H$ . The system consisting of the rules above and the rules from Figure 3 is sound for this semantics and complete for plain clauses.

LTL and CTL can express the semantics of annotated eventualities as a formula using the until operator. So given complete Hilbert systems for LTL [11] or CTL [7] it should be possible to translate derivations in Brännler and Lange’s calculi to Hilbert proofs. Unlike LTL and CTL,  $K^+$  cannot express the semantics of an annotated eventuality as a formula. Hence, it is not clear how to translate the rules above to Hilbert refutations in the Hilbert system.

The motivation for our Gentzen system was the need for a simple inductive characterization of unsatisfiability that can be translated to Hilbert refutations to constructively show the completeness of the Hilbert system. In fact, our monolithic compound rule allows us to directly read off the instantiation of the Segerberg axiom. So while for constructive completeness proofs for Hilbert systems for LTL and CTL history-based Gentzen systems seem promising, a system with a compound rule appears essential for weaker logics like  $K^+$ .

**Acknowledgments.** We thank Chad Brown for many helpful discussions and the suggestion to consider  $\Box^+ s$  rather than  $\Box^* s$  as primitive. We also thank the anonymous referees for their helpful comments.

## References

1. Ben-Ari, M., Pnueli, A., Manna, Z.: The temporal logic of branching time. *Acta Inf.* 20, 207–226 (1983)
2. Bertot, Y., Gonthier, G., Biha, S.O., Pasca, I.: Canonical Big Operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 86–101. Springer, Heidelberg (2008)

3. Brünnler, K., Lange, M.: Cut-free sequent systems for temporal logic. *J. Log. Algebr. Program.* 76(2), 216–225 (2008)
4. Doczkal, C., Smolka, G.: Coq formalization accompanying this paper, <http://www.ps.uni-saarland.de/extras/cpp12/>
5. Doczkal, C., Smolka, G.: Constructive Formalization of Hybrid Logic with Eventualities. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 5–20. Springer, Heidelberg (2011)
6. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Programming* 2(3), 241–266 (1982)
7. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. System Sci.* 30(1), 1–24 (1985)
8. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. *J. Comput. System Sci.*, 194–211 (1979)
9. Fitting, M.: Intuitionistic logic, model theory and forcing. *Studies in Logic.* North-Holland Pub. Co. (1969)
10. Fitting, M.: *Proof Methods for Modal and Intuitionistic Logics.* Reidel (1983)
11. Gabbay, D.M., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: Abrahams, P.W., Lipton, R.J., Bourne, S.R. (eds.) POPL, pp. 163–173. ACM Press (1980)
12. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009)
13. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A Modular Formalisation of Finite Group Theory. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 86–101. Springer, Heidelberg (2007)
14. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA (2008), <http://hal.inria.fr/inria-00258384/en/>
15. Harel, D., Kozen, D., Tiuryn, J.: *Dynamic Logic.* The MIT Press (2000)
16. Kaminski, M., Schneider, T., Smolka, G.: Correctness and Worst-Case Optimality of Pratt-Style Decision Procedures for Modal and Hybrid Logics. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNCS, vol. 6793, pp. 196–210. Springer, Heidelberg (2011)
17. Smullyan, R.M.: *First-Order Logic.* Springer (1968)
18. Sozeau, M.: A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning* 2(1) (2009)
19. The Coq Development Team, <http://coq.inria.fr>
20. Troelstra, A.S., Schwichtenberg, H.: *Basic proof theory*, 2nd edn. Cambridge University Press, New York (2000)



# Rating Disambiguation Errors<sup>\*</sup>

Andrea Asperti and Wilmer Ricciotti

Department of Computer Science, University of Bologna  
{`aspersi,ricciott`}@cs.unibo.it

**Abstract.** Ambiguous notation is a powerful tool developed to deal with the complexity of mathematics without sacrificing clarity or conciseness. In the mechanized parsing of ambiguous terms, a disambiguation algorithm can be used to provide the system with the intelligence necessary to select valid interpretations for the overloaded symbols received in input.

Disambiguation works by means of an incremental analysis of the input term, progressively discarding all invalid interpretations. As a result, if the input term cannot be disambiguated, many errors will be produced, only a handful of which are truly meaningful to the user.

In this paper, we improve the existing technique to classify disambiguation errors by introducing a new heuristic to sort errors from the most meaningful to the least, showing that it can be implemented in a natural way in the existing disambiguation algorithm. We also describe a neat interface to present disambiguation errors to the user, suitable for the use in interactive theorem proving applications.

## 1 Introduction

One of the most notable features of mathematical notation is ambiguity: for instance, it is possible to overload operators, as long as the intended interpretation of a given formula can be inferred from its context. On the other hand, results to be stored in formal libraries of interactive provers [13], need to be in an unambiguous form.

Ambiguous notation serves an important purpose, hiding redundant information and providing a standardized lexicon through which mathematicians can communicate more easily. It is therefore important that tools for mechanized mathematics be able to bridge the gap between mathematical notation and the unambiguous formalism used by the system. Hence, all interactive theorem provers address the issue of ambiguous notation in some way. Some provers try to resolve the ambiguity at parsing time by means of a deterministic system of interpretation scopes (an approach used in Coq). A more sophisticated technique popularized by the Haskell programming language and extended to theorem proving first in Isabelle, then in Coq and Matita, is that of *type classes* [12]; in this case every notation is associated unambiguously to a certain type class and

---

<sup>\*</sup> The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

parsed as such; each type class provides several possible overloaded instances of the notation, allowing the intended one to be selected later on during semantic analysis.

A different approach supported in Matita [3] since the very beginning, allows the parser to produce an ambiguous abstract syntax tree from ambiguous notation, which is later fed to a component called *disambiguator*, in charge of deriving all the possible well-typed interpretations. This paper is an ideal continuation of two works by Sacerdoti Coen and Zacchiroli about the implementation of disambiguators [9,10]. In particular, we focus on disambiguation errors and how to predict how much informative they are going to be for the user.

While the disambiguator approach has a great flexibility, it turns out that error reporting in this setting has an even more critical status than in type-checking. Usually the disambiguation process fails because of a single mistake by the user: in such a case, a human reader is often able to infer the intended meaning of the user input and spot the mistake; from the system perspective, however, each combination of the possible interpretations of ambiguous parts of the input will yield a different error. That is to say, when the disambiguation process fails, we may be left with dozens of failing interpretations and no clear way to recognize the interesting ones.

Our goal is to allow users to recover their intended interpretation from the heap of failures in order to understand what went wrong. We do this by providing a heuristic criterion to rate how likely an interpretation is to be the one intended by the user. This criterion is implemented as a straightforward addition to Sacerdoti Coen and Zacchiroli's efficient disambiguation algorithm.

The structure of the paper is as follows: Section 2 deals with the notion of disambiguation and provides definitions that will be used in the rest of the paper; Section 3 recalls the efficient disambiguation, which will serve as a basis for the rest of this work. Our contributions are described beginning in Section 4, where we discuss some drawbacks of a previous technique to detect so called *spurious* errors; Section 5 presents an entirely novel criterion providing a quantitative analysis of how much errors can be expected to be relevant and discusses an extension of the disambiguation algorithm yielding this finer classification; Section 6 presents a user interface for reporting disambiguation errors in an orderly way. Both the algorithm and the user interface have been implemented and are used as part of the Matita web application<sup>1</sup>.

## 2 The Notion of Disambiguation

In the handbook approach to compilation, the semantic analysis phase is in charge of associating to an abstract syntax tree (AST) at most one interpretation (or no interpretation at all in the case of a static semantic error). In the context of the formalization of mathematics, however, we are willing to allow the user to employ the standard, ambiguous mathematical syntax, with the maximum degree of flexibility. In this scenario, the ambiguity of concrete syntax is

<sup>1</sup> See <http://matita.cs.unibo.it/matitaweb.shtml>

transferred by the parser to an *ambiguous AST*, where by ‘ambiguous’ we mean that it admits more than one interpretation. The process associating an AST to the set of all its valid interpretations is called *disambiguation*. We will now make these concepts more formal.

In this discussion, we will provide an abstract presentation to avoid sticking to a specific syntax or formalism. We will call *AST* a tree built from primitive nodes in the set  $\mathbb{S}$ ; the set of ASTs will be denoted by  $\mathbb{A}$ . For every node in  $s \in \mathbb{S}$  there exists an associated *interpretation domain*  $\mathbb{D}_s$  which we will also regard as primitive.

A node in an AST can be either *ambiguous* or *disambiguated*. An ambiguous node is a bare primitive node (obtained by parsing ambiguous concrete syntax); a disambiguated node is a pair  $\langle s, d \rangle$  such that  $d$  is an interpretation in  $\mathbb{D}_s$ . Disambiguated nodes can be an intermediate product of disambiguation, but can also result from parsing of unambiguous concrete syntax (which in turn can be a deliberate choice of the user, or the refinement of ambiguous user syntax by means of disambiguation feedback). A given node may occur multiple times in an AST, therefore we will denote occurrences (i.e. positions in an AST) by  $n, n', \dots$ . The lookup operation returning the node at position  $n$  in the AST  $t$  is denoted  $t(n)$ .

We call an AST containing occurrences of ambiguous nodes an *ambiguous AST*, and an AST containing only disambiguated nodes an *unambiguous AST*. The set of unambiguous ASTs is denoted  $\overline{\mathbb{A}}$ . The set of the occurrences of ambiguous nodes in an AST  $t$  is called *domain* of  $t$  and denoted  $dom(t)$ . A substitution for an AST  $t$  is a finite partial map from the domain of  $t$  to disambiguated nodes, such that an occurrence of an ambiguous node  $s$  is mapped to a corresponding disambiguated node  $\langle s, d \rangle$ . The substitution map is lifted from nodes to ASTs in the obvious way. We say that an AST  $t'$  is an instance of another AST  $t$ , or equivalently that  $t$  is a generalization of  $t'$  (notation:  $t \preceq t'$ ) if there exists a substitution  $\sigma$  such that  $t' = t\sigma$ . The following property follows immediately.

**Lemma 1.**  $\preceq$  is a partial order relation.

Our intuition tells us that the semantics of an unambiguous AST is unique and that the semantics of an ambiguous AST is the union of the semantics of all its unambiguous instances. For our purposes, we can identify the set of semantics of ASTs with the set of unambiguous ASTs  $\overline{\mathbb{A}}$ . The semantics of ASTs is then formalized as follows:

**Definition 1.** The semantics of ASTs  $\llbracket \cdot \rrbracket : \mathbb{A} \rightarrow \wp(\overline{\mathbb{A}})$  is a function associating to any AST the set of all its unambiguous instances

$$\llbracket t \rrbracket = \{t' \in \overline{\mathbb{A}} : t \preceq t'\}$$

Since every unambiguous AST is the only instance of itself, according to the above definition, the semantics of an unambiguous AST is a singleton, regardless of the interpretations of its nodes, as expected. However, this definition is in a sense too loose to be of any use, because it says nothing about the coherence

of the interpretations we chose. The most obvious example of incoherence is ill-typedness: if our choices yield an ill-typed AST, that AST must be considered meaningless and thus discarded. In our abstract context, we do not employ any concrete notion of well-typedness, but rather we will assume the existence of an oracle  $\mathcal{R}$  deciding whether an AST is *valid* or not: the oracle will return  $\checkmark$  in the former case, and an informative error message otherwise.

**Definition 2.** *The disambiguation function  $\mathcal{D} : \mathbb{A} \rightarrow \wp(\overline{\mathbb{A}})$  maps any AST to the set of all its valid interpretations*

$$\mathcal{D}(t) = \{t' \in \llbracket t \rrbracket : \mathcal{R}(t') = \checkmark\}$$

A trivial algorithm implementing  $\mathcal{D}$  consists of computing the set of all ground instances of the AST to be disambiguated and then filtering through the oracle  $\mathcal{R}$ . This technique is clearly inefficient, since the number of ground instances of an AST is exponential in the number of its ambiguous nodes.

### 3 A Disambiguation Algorithm

Efficient implementations of disambiguation operate by incrementally instantiating the original AST, immediately pruning those partial instances which can already be shown to be invalid by the oracle, and iterating the process until no ambiguous nodes are left. Early pruning leads to enormous performance improvements.

In order for this kind of implementation to work, we must relax the definition of  $\mathcal{R}$  to allow it to take ambiguous ASTs as input too. In this case, we want  $\mathcal{R}$  to always return  $\checkmark$  if the input AST can be instantiated to a valid unambiguous AST, because only invalid instances should be pruned. When this condition is satisfied, we clearly want as many ASTs as possible to be rejected, to minimize the number of incremental instantiations.

In summary, we would like the oracle to return an error if and only if all the instances of the input AST are invalid. However, in general we are not able to identify all the ambiguous ASTs not admitting valid instantiations: this happens for two reasons:

- while some errors are located in disambiguated parts of the AST and can be immediately recognized as such, other errors, located in ambiguous parts, can only be recognized after disambiguation of some nodes: to detect such errors, it is necessary to first instantiate some ambiguous nodes, making an efficient implementation of  $\mathcal{R}$  impossible;
- in the case where all instances of the input AST are invalid,  $\mathcal{R}$  should return a single message explaining why all such instances cannot be accepted: unfortunately, it may be the case that all the instances are invalid, but each of them is invalid for a different reason.

Our disambiguation algorithm will therefore assume that ASTs rejected by  $\mathcal{R}$  are invalid for all possible instantiations, but the inverse implication will not hold in general.

*Property 1.* Given a (possibly ambiguous) AST  $t$ , if  $\mathcal{R}(t)$  returns an error, then all instances of  $t$  are not valid.

*Property 2.* If  $\mathcal{R}(t)$  returns an error, that error is meaningful for all instances of  $t$ .

Given a set of ASTs  $\Sigma$ , we define the notations  $\Sigma^\checkmark$  and  $\Sigma^\times$  as follows:

$$\begin{aligned}\Sigma^\checkmark &= \{t \mid t \in \Sigma \wedge \mathcal{R}(t) = \checkmark\} \\ \Sigma^\times &= \{t, \mathcal{R}(t) \mid t \in \Sigma \wedge \mathcal{R}(t) \neq \checkmark\}\end{aligned}$$

Therefore,  $\Sigma^\checkmark$  will contain the subset of all the ASTs that are still valid, while  $\Sigma^\times$  will contain all the invalid ASTs in  $\Sigma$  paired with their error messages.

We can now show the “efficient” disambiguation algorithm originally presented in [9]. It follows the aforementioned criterion of incremental instantiation of the input AST.

```
procedure disambiguate( $t$ )
begin
   $\Sigma \leftarrow \{t\}^\checkmark$ ;  $\Omega \leftarrow \{t\}^\times$ ;
  while ( $\Sigma \neq \emptyset \wedge \text{next}(\Sigma) \neq \times$ )
    begin
       $n \leftarrow \text{next}(\Sigma)$ ;
       $\Delta \leftarrow \{u[n \mapsto \langle u(n), d \rangle] \mid u \in \Sigma, d \in \mathbb{D}_{u(n)}\}$ ;
       $\Sigma \leftarrow \Delta^\checkmark$ ;  $\Omega \leftarrow \Omega \cup \Delta^\times$ ;
    end
  return  $\Sigma, \Omega$ 
end
```

The algorithm maintains a set  $\Sigma$  of partially disambiguated instances of the input AST  $t$ , ensuring that they share the same domain and have not been rejected by  $\mathcal{R}$  yet ( $\Sigma$  is initialized as the singleton  $\{t\}$ , except when  $t$  is immediately rejected by  $\mathcal{R}$ , in which case it is initialized as the empty set, leading to failure). The algorithm is parametric on a procedure `next` taking as input a set of ASTs sharing the same domain and returning an element of that domain (an ambiguous node occurrence), or  $\times$  if no ambiguous node is left. In the **while** cycle, we choose a node `next`( $\Sigma$ ) from the domain of  $\Sigma$  and instantiate it in all possible ways, obtaining a new set  $\Delta$ . We then filter out invalid instances obtaining a new set  $\Sigma$  to continue iteration. The cycle stops when either all ambiguous nodes have been instantiated (`next`( $\Sigma$ ) =  $\times$ ), meaning the disambiguation was successful, or when  $\Sigma$  is empty, meaning all the instances of  $t$  are invalid.  $\Omega$  is used to collect all the errors produced by  $\mathcal{R}$ .

To discuss the properties of the algorithm, we introduce a notation to refer to the value of a variable at a specific iteration of a while cycle: we will use  $v_0$  to denote the value of a variable  $v$  before the first iteration, and  $v_i$  to denote its value at the end of the  $i$ -th iteration.

**Lemma 2.** *In an execution of `disambiguate`( $t$ ), for all  $i \geq 1$ , each AST  $t'$  in  $\Delta_i$  is such that  $|\text{dom}(t')| = |\text{dom}(t)| - i$*

*Proof.* By induction on  $i$ : when  $i = 1$ , we instantiate a single ambiguous node from the original AST  $t$ , and the thesis follows easily; when  $i > 1$ , we know that  $\Sigma_{i-1}$  is a subset of  $\Delta_{i-1}$ , so by induction hypothesis every AST in it has a domain of cardinality  $|\text{dom}(t)| - i + 1$ : to compute  $\Delta_i$ , we instantiate one ambiguous node more, therefore getting a domain of cardinality  $\text{dom}(t) - i$ , as needed.

**Lemma 3.** *The `disambiguate` algorithm terminates after at most  $|\text{dom}(t)|$  executions of the **while** cycle.*

*Proof.* Trivial, since at each iteration, either  $\Sigma$  becomes empty (and the algorithm terminates immediately), or the number of ambiguous nodes in  $\Sigma$  decreases by one (easily proved by means of Lemma 2), eventually reaching 0 and falsifying the guard of the **while** cycle.

**Lemma 4.** *Given an AST  $t$ , for all  $t' \in \overline{\mathbb{A}}$  such that  $t \preceq t'$  and for all  $i \leq |\text{dom}(t)|$ , there exists  $t''$  such that  $t \preceq t'' \preceq t'$  and either  $t'' \in \Sigma_i$  and  $\mathcal{R}(t'') = \checkmark$ , or  $t'' \in \Omega_i$  and  $\mathcal{R}(t'') \neq \checkmark$ .*

*Proof.* We proceed by induction on  $i$ . If  $i = 0$ , then we choose  $t'' = t$  and the statement is satisfied. If  $i > 0$ , by induction hypothesis there exists  $t'''$  such that  $t \preceq t''' \preceq t'$  and either  $\mathcal{R}(t''') = \checkmark$  and  $t''' \in \Sigma_{i-1}$  or  $\mathcal{R}(t''') \neq \checkmark$  and  $t''' \in \Omega_{i-1}$ . If  $t''' \in \Omega_{i-1}$ , we choose  $t'' = t'''$  and get the thesis since  $\Omega_{i-1} \subseteq \Omega_i$ . If  $t''' \in \Sigma_{i-1}$ , we choose  $t'' = t'''[n_i \mapsto t'(n_i)]$ : clearly  $t \preceq t'' \preceq t'$  by definition; furthermore,  $t'' \in \Delta_i$ . If  $\mathcal{R}(t'') = \checkmark$ , then we proved that  $t'' \in \Sigma_i$ ; otherwise,  $t'' \in \Omega_i$ . In both cases, the thesis holds.

The two following theorems assert the soundness of the algorithm, respectively saying that the  $\Sigma$  returned by the algorithm is the set of all valid disambiguated instances of the input, and that all invalid disambiguated instances of the input have an invalid generalization in the  $\Omega$  returned by the algorithm (or equivalently, that  $\Omega$  contains an error explaining why that AST is invalid).

**Theorem 1.** *The set  $\Sigma$  returned by `disambiguate`( $t$ ) is equal to  $\mathcal{D}(t)$ .*

*Proof.* We prove that the algorithm returns a  $\Sigma$  such that  $\Sigma \subseteq \mathcal{D}(t)$  and  $\Sigma \supseteq \mathcal{D}(t)$ .

$\Sigma \subseteq \mathcal{D}(t)$ : Since only valid ASTs ever enter  $\Sigma$  and the cycle only terminates when the domain of ASTs in  $\Sigma$  is empty (or  $\Sigma = \emptyset$ ),  $\Sigma$  only contains unambiguous valid ASTs, thus  $\Sigma \subseteq \mathcal{D}(t)$ .

$\Sigma \supseteq \mathcal{D}(t)$ : By lemmata 3 and 4, we can prove that at the last execution of the **while** cycle, for all  $t' \in \mathcal{D}(t)$ , there exists  $t'' \preceq t'$  such that  $t'' \in \Sigma$ ; this implies  $\Sigma$  is not empty and  $t''$  is unambiguous (otherwise, the cycle would execute another time). It thus follows that  $t'' = t'$  and, consequently,  $t' \in \Sigma$ .

**Theorem 2.** *Let  $t$  be an AST and  $\Omega$  the error collection returned by `disambiguate`( $t$ ). Then:*

1. given an AST  $t'$  and an error message  $e$  such that  $\langle t', e \rangle \in \Omega$ , for all  $t''$  such that  $t' \preceq t''$  we have  $\mathcal{R}(t'') \neq \checkmark$ ;
2. for all  $t'$  such that  $t \preceq t'$  and  $t' \notin \mathcal{D}(t)$ , there exists an AST  $t''$  such that  $t \preceq t'' \preceq t'$  and  $\langle t'', \mathcal{R}(t'') \rangle \in \Omega$ .

*Proof.* Part 1 is trivial (only invalid ASTs enter  $\Omega$ , and by Property 1 of  $\mathcal{R}$ , all their instances must also be invalid).

To prove part 2, let  $k$  be the number of iterations after which the algorithm terminates. We have  $\Sigma = \Sigma_k$  and  $\Omega = \Omega_k$ : thus by using Lemma 4 with  $i = k$ , we get a  $t''$  such that  $t \preceq t'' \preceq t'$  and either  $t'' \in \Sigma$  or  $t'' \in \Omega$ . In the first case by Theorem 1,  $t'' \in \mathcal{D}(t)$ . This implies  $t''$  is also unambiguous, thus from  $t'' \preceq t'$  we also get  $t'' = t'$ . But then  $t' \in \mathcal{D}(t)$ , which falsifies our hypothesis. If instead  $t'' \in \Omega$ , the thesis follows immediately.

The choice of `next` influences both efficiency and the errors returned by the algorithm. Since in general the interpretation of one node constrains the interpretation of all its children, while the constraints imposed by a node to its parent and siblings are much less restrictive (if they exist at all), and since, as we noted, the constraints we imposed on the validity test are such that no reasonable implementation is allowed to consider the children of an ambiguous node, the `next` function should be implemented by visiting the nodes nearest to the root first, as in a pre-order or level-order (breadth first) traversal.

## 4 Spurious Errors

When disambiguation is successful, it is generally going to return a small set of choices (most usually, just one), all of which are meaningful: in this case, the errors produced during the disambiguation process can be ignored. If disambiguation fails, however, we want to provide the user with information on what went wrong, by returning to him the  $\Omega$  set computed by `disambiguate`. This set can easily contain a large amount of invalid interpretations that to the system are equally wrong, even though the user is likely to have committed just one mistake; the vast majority of errors produced by the disambiguation algorithm is *spurious*: they do not correspond to a user error, but are only a technical means to drive the disambiguation algorithm to the correct interpretation (if it exists).

In order to provide the user with more accurate information, a heuristic criterion to distinguish genuine errors from spurious errors was introduced in [10]:

**Criterion 1 (Spurious Error Detection).** *An error is spurious when it is localized in a sub-formula  $F$  such that there is an alternative interpretation of the formula such that no error is located in  $F$ .*

The meaning of the criterion is clear: the system should try to interpret the input AST as much as possible and keep as real errors only those that are “unrecoverable”, i.e. those for which no alternative valid interpretation exists.

However Criterion 1 lacks a clear implementation, especially because an efficient algorithm (like the one presented in the previous section) does not consider all possible interpretations (and consequently all possible errors) of the input AST. For this reason, the following restriction of the criterion 2, allowing a more obvious implementation, has been suggested in the aforementioned paper (here rephrased to make it agree with our simpler presentation of Section 2):

**Criterion 2 (Draconian Spurious Error Detection).** *During the disambiguation of  $t$ , an error message relative to an instance  $t'$  of  $t$  is spurious iff there exists an occurrence  $n \in \text{dom}(t)$  and an alternative instance  $t''$  of  $t$  such that:*

1.  $t'(n) \neq t''(n)$ ;
2.  $t', t''$  are both unambiguous on all  $n'$  preceding  $n$  in pre-order;
3.  $\mathcal{R}(t'') = \checkmark$ ;

Let us point out the heuristic status of the criterion: the causal relation between the interpretation of a node and an error is informal and cannot be grasped accurately by any disambiguation algorithm. In practice, this means that we may sometimes find genuine errors classified as spurious.

A second suspicious point is the second requirement of the criterion, explicitly stating the traversal algorithm to be used in the disambiguation process. The choice of a pre-order traversal is sensible but arbitrary (a breadth-first traversal also enjoys good properties with respect to the disambiguation algorithm). Indeed, the classification of spurious errors is dependent on the order in which subterms are considered: in some cases, opposite classifications can be performed on semantically equivalent terms, differing only by a commutativity property. The next example shows this anomaly.

*Example 1.* Consider the concrete syntax

$$(\alpha + \beta) + (\alpha + \gamma) = (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$$

where  $\alpha, \beta, \gamma$  are interpreted to be in  $\mathbb{R}$  and  $\mathbf{x}, \mathbf{y}, \mathbf{z}$  in  $\mathbb{R}^2$ . Assume that the interpretation domain for  $+$  contains elements `plusR` and `plusV` representing respectively sum on real scalars and vectors; similarly, the domain for  $=$  will contain interpretations `eqR` and `eqV` for equality on scalars and vectors. In a disambiguation algorithm classifying spurious errors, the nodes will be considered and assigned interpretations in the pre-order sequence; disambiguation of the above formula will fail because the two sides of the equality have different types. Immediately before failure, the only instance of the original AST still being processed is

$$(\alpha +_{\text{plusR}} \beta) +_{\text{plusR}} (\alpha +_{\text{plusR}} \gamma) =_{\text{eqR}} (\underline{\mathbf{x} + \mathbf{y}}) +_{\text{plusR}} (\mathbf{x} + \mathbf{z})$$

where the symbol being considered is the underlined one. Both interpretations in its domain will fail, returning errors:

<sup>2</sup> The criterion is called *draconian* because it recognizes as spurious more errors than the *prudent* criterion also proposed in [10]. It is not possible to discuss both the criteria here due to space constraints, but the considerations we are going to draw apply to the prudent criterion as well.



- $(\alpha +_{\text{plusR}} \beta) +_{\text{plusR}} (\alpha +_{\text{plusR}} \gamma) =_{\text{eqR}} (\mathbf{x} +_{\text{plusR}} \mathbf{y}) +_{\text{plusR}} (\mathbf{x} + \mathbf{z})$ :  $\mathbf{x}$  has type vector but is here used as a scalar
- $(\alpha +_{\text{plusR}} \beta) +_{\text{plusR}} (\alpha +_{\text{plusR}} \gamma) =_{\text{eqR}} (\mathbf{x} +_{\text{plusV}} \mathbf{y}) +_{\text{plusR}} (\mathbf{x} + \mathbf{z})$ :  $\mathbf{x} +_{\text{plusV}} \mathbf{y}$  has type vector but is here used as a scalar

After generating those errors, no valid interpretation is left and the algorithm will stop. Errors produced by the disambiguation of preceding nodes in the pre-order sequence will be flagged as spurious by the draconian criterion, including the errors precluding the `plusV` interpretation for  $\alpha + \beta$ :

- $(\alpha +_{\text{plusR}} \beta) +_{\text{plusV}} (\alpha + \gamma) =_{\text{eqV}} (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$ :  $\alpha +_{\text{plusR}} \beta$  has type scalar but is here used as a vector
- $(\alpha +_{\text{plusV}} \beta) +_{\text{plusR}} (\alpha + \gamma) =_{\text{eqR}} (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$ :  $\alpha$  has type scalar but is here used as a vector

If on the contrary we consider the symmetric equation

$$(\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z}) = (\alpha + \beta) + (\alpha + \gamma)$$

the disambiguation will proceed until the only interpretation left is:

$$(\mathbf{x} +_{\text{plusV}} \mathbf{y}) +_{\text{plusV}} (\mathbf{x} +_{\text{plusV}} \mathbf{z}) =_{\text{eqV}} (\alpha +_{\text{plusR}} \beta) +_{\text{plusV}} (\alpha + \gamma)$$

The system will then return the following errors as meaningful:

- $(\mathbf{x} +_{\text{plusV}} \mathbf{y}) +_{\text{plusV}} (\mathbf{x} +_{\text{plusV}} \mathbf{z}) =_{\text{eqV}} (\alpha +_{\text{plusR}} \beta) +_{\text{plusV}} (\alpha + \gamma)$ :  $\alpha +_{\text{plusR}} \beta$  has type scalar but is here used as a vector
- $(\mathbf{x} +_{\text{plusV}} \mathbf{y}) +_{\text{plusV}} (\mathbf{x} +_{\text{plusV}} \mathbf{z}) =_{\text{eqV}} (\alpha +_{\text{plusV}} \beta) +_{\text{plusV}} (\alpha + \gamma)$ :  $\alpha$  has type scalar but is here used as a vector

In this case, the errors about  $\mathbf{x} + \mathbf{y}$  will be flagged as spurious, showing that the notion of spuriousness is not stable under minor syntactic modifications of the input. Arguably, in both versions of the equation,  $\mathbf{x} + \mathbf{y}$  and  $\alpha + \beta$  are equally wrong (since they are both responsible for the whole equation being rejected).

## 5 Error Rating

Example [□](#) shows that in some cases an error is classified as spurious only because of its position in the formula to be disambiguated, even though a user would recognize it as a real error. We attribute this anomaly to the extreme coarseness of the distinction spurious/non-spurious: if we could establish a *rating* criterion capable of distinguishing more than two degrees of significance, it would be possible to present errors to the user so that the most meaningful come first, followed by the less meaningful in a gradual fashion.

Our intent is therefore to understand what are the features of an error that is meaningful to the user. Typically, a meaningful error tells the user something interesting by contrasting large valid subterms with a single incoherent node; according to this point of view, an erroneous AST should be rated depending on

its valid generalizations. Let us call *maximal valid generalization* of a (possibly invalid) AST  $t$  an AST  $t'$  that is a valid generalization of  $t$  and such that all other valid generalizations of  $t$  have fewer interpreted nodes than  $t'$ . The more nodes are interpreted by  $t'$ , the better the rating of  $t$  should be.

Essentially, when rating erroneous interpretations, we want to privilege those that are closer to being valid because their maximal valid generalizations have more interpreted nodes. This requirement is expressed by the following criterion.

**Criterion 3 (Error rating criterion).** *Given two erroneous partial instances  $t_1$  and  $t_2$  of the same input AST  $t$ , the error for  $t_1$  is less likely than the error in  $t_2$  (notation:  $t_1 \preceq t_2$ ) iff there exists a valid generalization of  $t_2$  whose domain is smaller than the domain of all valid generalizations of  $t_1$ . Formally:*

$$t_1 \preceq t_2 \iff \left( \exists t'_2 \preceq t_2 : \mathcal{R}(t'_2) = \checkmark \wedge \forall t'_1 \preceq t_1 : \mathcal{R}(t'_1) = \checkmark \implies |dom(t'_1)| \geq |dom(t'_2)| \right)$$

We can also express the rating of an AST by means of a natural number using the following rating function.

**Definition 3 (Rating function).** *The rating of an AST  $t$  (notation:  $\varrho(t)$ ) is defined as the smallest cardinality of the domains of all its valid generalizations. Formally:*

$$\varrho(t) \triangleq \min_{t' \preceq t \wedge \mathcal{R}(t') = \checkmark} |dom(t')|$$

According to this definition, the lower the rating, the more likely an AST is to be what the user originally intended. In particular, a valid unambiguous AST receives a rating of 0.

**Lemma 5.** *For all  $t_1, t_2$ ,  $t_1 \preceq t_2$  iff  $\varrho(t_1) \geq \varrho(t_2)$ .*

*Proof.*

$\implies$ : By the definition of likelihood according to Criterion 3, there exists a valid generalization  $t'_2$  of  $t_2$  such that for all valid generalizations  $t'_1$  of  $t_1$ ,  $|dom(t'_1)| \geq |dom(t'_2)|$ ; on the other hand, the definition of  $\varrho$  implies that  $\varrho(t_1) = |dom(t'_1)|$  for some  $t'_1$ . By taking  $t'_1 = t'_1$ , we have  $\varrho(t_1) = |dom(t'_1)| \geq |dom(t'_2)|$ , and by the definition of  $\varrho$ ,  $|dom(t'_2)| \geq \varrho(t_2)$ . Then the thesis holds by transitivity of  $\geq$ .

$\impliedby$ : By the definition of  $\varrho$ , let  $t'_2$  be a valid generalization of  $t_2$  such that  $|dom(t'_2)| = \varrho(t_2)$ . Then, again by definition of  $\varrho$ , we know that for all valid generalizations  $t'_1$  of  $t_1$ ,  $|dom(t'_1)| \geq \varrho(t_1)$ ; then combining the hypothesis we get  $|dom(t'_1)| \geq |dom(t'_2)|$ ; by the definition of likelihood, this yields the thesis.

**Corollary 1.**  $\preceq$  is a total order relation.

*Proof.* A consequence of  $\geq$  being a total order relation, by means of Lemma 5.

The rating of an AST provides a formal, yet very natural way of assessing the significance of an interpretation, even when it is not valid. Anyway, the definition

we gave does not provide an immediate method for computing the rating of an AST: enumerating the generalizations of a given AST until a valid one is found could be computationally expensive. Luckily, it is possible to generalize the efficient disambiguation algorithm so that it returns errors sorted depending on their rating.

```

procedure disambiguate_and_rate( $t$ )
begin
   $\Sigma \leftarrow \{t\}^\checkmark$ ;
  if  $\Sigma \neq \emptyset$  then  $\Omega \leftarrow []$  else  $\Omega \leftarrow [\{t\}^\times]$ ;
  while ( $\Sigma \neq \emptyset \wedge \text{next}(\Sigma) \neq \times$ )
    begin
       $n \leftarrow \text{next}(\Sigma)$ ;
       $\Delta \leftarrow \{u[n \mapsto \langle u(n), d \rangle] \mid u \in \Sigma, d \in \mathbb{D}_{u(n)}\}$ ;
       $\Sigma \leftarrow \Delta^\checkmark$ ;
      if  $\Delta^\times \neq \emptyset$  then  $\Omega \leftarrow \Delta^\times :: \Omega$ ;
    end
  return  $\Sigma, \Omega$ 
end

```

It is easy to show that the  $\Sigma$  and  $\Omega$  returned by `disambiguate_and_rate` contain exactly the same ASTs as those returned by `disambiguate`, thus the soundness of this algorithm descends from that of the other one. The whole difference between the two lies not in the content of  $\Omega$ , but in its structure. First it is not a set anymore, but a list of sets; each element in the list, which we will call *error frame*, is obtained from the failing interpretations in a certain  $\Delta_i$ : thus all the failing interpretations have the same domain, and are failing after the instantiation of the same node. This allows us to prove the following theorem.

**Theorem 3.** *The list  $\Omega$  returned by `disambiguate_and_rate` is sorted by decreasing likelihood, that is, if  $\Omega = [\omega^1, \omega^2, \dots, \omega^m]$ , then for all  $t_i \in \omega^i$  and  $t_j \in \omega^j$  where  $i \leq j$ ,  $t_j \preceq t_i$ .*

*Proof.* It is easy to prove that  $\Omega = [\Delta_{k_1}^\times, \Delta_{k_2}^\times, \dots, \Delta_{k_m}^\times]$ , such that  $k_i > k_j$  iff  $i < j$ . Therefore, we will prove that if  $i < j$ , then  $t_i \in \Delta_{k_i}^\times$  and  $t_j \in \Delta_{k_j}^\times$  are such that  $t_j \preceq t_i$ . We know from the definition of the algorithm that for all  $h$ , each  $t' \in \Delta_h$  is obtained from some AST  $t''$  in  $\Sigma_{h-1}$  by instantiating a single ambiguous node, and that each AST in  $\Sigma_{h-1}$  is valid. This implies  $\varrho(t_i) = |\text{dom}(t_i)| + 1$  and  $\varrho(t_j) = |\text{dom}(t_j)| + 1$ . By Lemma 2, we prove that  $|\text{dom}(t_i)| = |\text{dom}(t)| - k_i$  and  $|\text{dom}(t_j)| = |\text{dom}(t)| - k_j$ ; since  $k_i > k_j$ , we prove that  $\varrho(t_j) \geq \varrho(t_i)$  and by Lemma 5,  $t_j \preceq t_i$ .

We still have to decide whether we should prefer, as the implementation of `next`, an in-order visit or a level-order visit. The issue cannot be addressed exclusively on the basis of efficiency, since it is easy to show ASTs on which the former choice outperforms the latter, and vice-versa. However our tests indicate that our algorithm behaves better when employed with a level-order visit, in the sense

that the error ordering produced is closer to the expected one. This is probably due to the fact that the interpretations of node occurrences located nearer to the leaves (and consequently the related error messages) tend to be more significant from the user point of view than those of nodes located nearer to the root of the AST. An in-order visit, on the other hand, alternates deep and shallow node occurrences and should therefore be avoided.

## 6 Error Reporting

An attractive option for reporting disambiguation errors is to let the user disambiguate manually individual error locations by means of a point-and-click interface, until the amount of disambiguation errors and their quality is judged to be satisfying. Our original intention was to implement such an interface; however our experience as users of interactive theorem provers tells us that the user is frequently incapable of guessing where disambiguation went wrong. The reason is that the combination of overloading with other advanced features of theorem provers, especially dependent types and coercive subtyping, make the disambiguation process quite intricate from a human perspective. Since the user is reviewing errors precisely for the purpose of understanding what went wrong, it is highly likely that a single interaction at a random error location with a point-and-click interface will not be clarifying at all, leading to frustration.

Our rating algorithm was designed having in mind that in this quite unusual case, the system knows better than the user which errors are the best candidates to being genuine, thanks to the rating criterion we proposed. We will employ it to design a user interface abiding by the following requirements:

- errors should be grouped according to their location;
- users should see those errors that are more meaningful first;
- the number of errors shown at the same time should be manageable.

Classification of spurious errors in the style of Section 4 only partially respects these requirements: in particular, errors categorized as non-spurious respect the requirements, but the other ones do not. As we saw in Example 1, some errors that are morally non-spurious may be categorized as spurious too, meaning they will be intermingled with maybe dozens of uninteresting errors coming from mixed locations.

On the other hand, the  $\omega^i$  sets in the list  $\Omega$  returned by our algorithm seem to be good candidates for the use in an interface satisfying the aforementioned requirements. All the errors in the same set were produced at the same location in the AST, thus satisfying the first requirement; the ordering of the list  $\Omega$  asserted by Theorem 3 provides a good basis for respecting the second requirement; finally, partitioning the errors in possibly more than just two sets (spurious and non-spurious) guarantees that our algorithm will perform better also with respect to the last requirement.

Each frame  $\omega^i$  is obtained by interpreting one ambiguous node occurrence of all ASTs in  $\Sigma$  in all possible ways, filtering by means of  $\mathcal{R}$  and keeping only the invalid instances. This has two consequences:

- (a) not all interpretations possible for the node may be present in the frame, because some of them may only be shown to be invalid after the AST is instantiated further;
- (b) some interpretations for the node may yield more than one invalid AST (in particular, up to one for each AST in  $\Sigma$ ).

Given an interpretation for the node occurrence being considered in  $\omega^i$ , we call the subset of ASTs instantiated with that interpretation a *slice* of  $\omega^i$ . All the ASTs in the same slice have the same interpretation for the last node considered occurrence, but differ in the interpretation of at least another node.

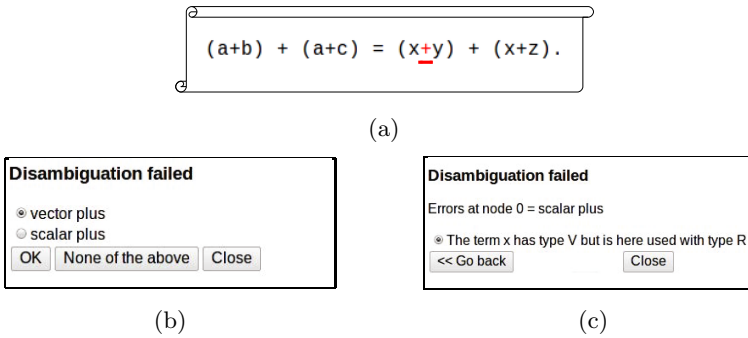


Fig. 1. Error reporting interface

We propose a user interface composed of two panes. The first pane, called *interpretation pane* (Figure 1(b)), shows, for a given frame, a list of the interpretations of the node occurrence being considered yielding a disambiguation error; we use highlighting directly in the user input (Figure 1(a)) to show the node currently being considered. After choosing an interpretation in the interpretation pane, we are sent to an *error list pane* (Figure 1(c)), reporting the list of all the errors associated with that choice (i.e. a slice of the frame).

The activity diagram in Figure 2 shows the intended user interaction with this interface. The user will initially be shown a list of interpretations from the frame  $\omega^1$  (which is the most likely to contain meaningful errors): if the user intended the current node to be associated to one of those interpretations, he will choose it and immediately view the list of related errors; otherwise, he will use the *None of the above* button to switch the view to the following frame and iterate the procedure. After viewing the error list, the user can either be satisfied with the errors shown (when at least one of them explains what went wrong), or decide to go back to the interpretation pane to try selecting a different interpretation or inspecting another frame.

*Example 2.* Consider again the AST for the expression

$$(\alpha + \beta) + (\alpha + \gamma) = (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$$

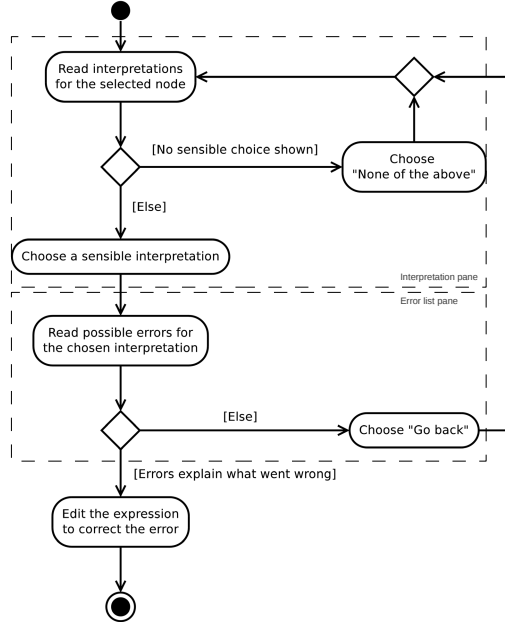


Fig. 2. User interaction with the error reporting interface

from Example 1, using for the variables the same types as we used in that example. After using the `disambiguate_and_rate` algorithm with a level-order traversal of the AST, disambiguation will fail and the user interface will highlight a symbol in the original expression:

$$(\alpha + \beta) + (\alpha + \gamma) = (\mathbf{x} \underline{+} \mathbf{y}) + (\mathbf{x} + \mathbf{z})$$

Among the two possibilities, in the interpretation pane we choose “vector plus”. The interface returns a single error:

- the term  $\mathbf{x} +_{\text{plusV}} \mathbf{y}$  has type vector but is here used with type scalar.

We decide that the error is not informative to us: something went wrong in a different part of the AST, therefore we switch to the next error frame.

$$(\alpha + \beta) + (\alpha \underline{+} \gamma) = (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$$

This time, the interpretation pane only shows the choice “vector plus”, which is not the intended one. We switch again to the next error frame:

$$(\alpha \underline{+} \beta) + (\alpha + \gamma) = (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$$

After choosing the interpretation “scalar plus”, the error list pane will show us the error

- the term  $\alpha +_{\text{plusR}} \beta$  has type scalar but is here used with type vector.

Now we realize that the expression we intended to write was

$$f(\alpha + \beta) + f(\alpha + \gamma) = (\mathbf{x} + \mathbf{y}) + (\mathbf{x} + \mathbf{z})$$

for a given function  $f$  from scalars to vectors. As we noted in Example 11, if we had used the spurious error classification, the error message about  $\alpha + \beta$  would have been considered spurious, making it much more difficult to spot it.

## 7 Conclusions

There have been considerable research efforts devoted to improving the way type errors are reported to the user. Most works are concerned with type systems à la Hindley-Milner, whose type-inference algorithm has been shown to work in a way that is substantially different from how people commonly reason about types. Such works (among which we mention those by Jun, Michaelson, and Trinder [6], Hage and Heeren [5], and Stuckey, Sulzmann and Wasny [11]) are mainly interested in improving the error message that is returned to the user by means of several heuristics. Another relevant proposal by Rittri ([8]) is devoted to the design of an interactive interface that can help explain to the user the source of a type error.

Both kinds of work bear some resemblance to our implementation, in the spirit if not in the letter. Given the fact that the notion of disambiguation error is more general than that of type error, to improve the user experience we are urged to address a different kind of problem: how to help the user to discriminate between genuine errors and spurious errors.

We do this by means of a new algorithm that is capable of partitioning and sorting errors according to their significance. This constitutes a remarkable improvement over the previous technique of spurious error detection, which is only capable of distinguishing two degrees of significance. In addition to this, we also believe that our approach is based on a more understandable principle, which does not involve implementation details such as the order in which nodes in an AST are visited.

Even though the two approaches stem from different analyses of the problem, the solutions have more in common than expected: it can easily be shown that, when an in-order traversal is chosen for the `disambiguate_and_rate` algorithm, the error list returned by it is structured in such a way that the topmost frame contains the genuine errors and the rest of the list contains the spurious ones (according to the draconian criterion). In this sense, the rating of disambiguation errors is a refinement of the discrimination of spurious errors.

We kept our discussion considerably abstract, making only a few weak and plausible assumptions on the structure of ASTs and on the existence of a validity test  $\mathcal{R}$ ; this allows our algorithm and interface to be used in a wide range of applications, including of course interactive theorem provers. In particular, in our implementation of the disambiguation algorithm in Matita,  $\mathcal{R}$  operates by first translating the input AST to a term in the foundational language of the system – a variant of the Calculus of (Co)Inductive Constructions extended with

metavariables in the style of [47] – in such a way that ambiguous nodes and their descendants are replaced by fresh metavariables (this ensures that Property 1 of Section 3 holds). The obtained term is then fed to the refinement facility of the system [2] for a typability check.

Our final remarks are about the user experience. The new disambiguation infrastructure has been developed recently as part of the web application version of Matita [1]. Our impression is that it provides a marked improvement over the past, especially because the interface is much less invasive. Due to this change being so new, there could still be room for improvement and we are committed to considering opinions and suggestions coming from the users of the system.

## References

1. Asperti, A., Ricciotti, W.: A Web Interface for Matita. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS, vol. 7362, pp. 417–421. Springer, Heidelberg (2012)
2. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *LMCS* 8(1) (2012)
3. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita Interactive Theorem Prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)
4. Geuvers, H., Jojgov, G.I.: Open Proofs and Open Terms: A Basis for Interactive Logic. In: Bradfield, J.C. (ed.) CSL 2002. LNCS, vol. 2471, pp. 537–552. Springer, Heidelberg (2002)
5. Hage, J., Heeren, B.: Heuristics for Type Error Discovery and Recovery. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 199–216. Springer, Heidelberg (2007)
6. Jun, Y., Michaelson, G., Trinder, P.: Explaining polymorphic types. *Comput. J.* 45(4), 436–452 (2002)
7. Muñoz, C.: A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory. PhD thesis, INRIA (November 1997)
8. Rittiri, M.: Finding the source of type errors interactively. Technical report, Department of Computer Science, Chalmers University of Technology, Sweden (1993)
9. Sacerdoti Coen, C., Zacchiroli, S.: Efficient Ambiguous Parsing of Mathematical Formulae. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 347–362. Springer, Heidelberg (2004)
10. Sacerdoti Coen, C., Zacchiroli, S.: Spurious disambiguation errors and how to get rid of them. *Journal of Mathematics in Computer Science, Special Issue on MKM 2*, 355–378 (2008)
11. Stuckey, P.J., Sulzmann, M., Wazny, J.: Improving type error diagnosis. In: Proceedings of 2004 ACM SIGPLAN Haskell Workshop, Haskell 2004, pp. 80–91. ACM, New York (2004)
12. Wenzel, M.: Type Classes and Overloading in Higher-Order Logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997)
13. Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)



# A Formal Proof of Square Root and Division Elimination in Embedded Programs

Pierre Neron

École Polytechnique - INRIA  
pierre.neron@inria.fr

**Abstract.** The use of real numbers in a program can introduce differences between the expected and the actual behavior of the program, due to the finite representation of these numbers. Therefore, one may want to define programs using real numbers such that this difference vanishes. This paper defines a program transformation for a certain class of programs that improves the accuracy of the computations on real number representations by removing the square root and division operations from the original program in order to enable exact computation with addition, multiplication and subtraction. This transformation is meant to be used on embedded systems, therefore the produced programs have to respect constraints relative to this kind of code. In order to ensure that the transformation is correct, *i.e.*, preserves the semantics, we also aim at specifying and proving this transformation using the PVS proof assistant.

**Keywords:** Program Transformation, Program Verification, Real Number Arithmetic, Safety Critical Embedded Systems, Semantics.

Safety critical embedded systems, for instance in aeronautics, demand a very high level of reliability since any failure can have critical consequences. A good way to achieve such a level of reliability is to prove properties on these programs using proof assistants such as PVS, COQ, HOL,... The programs we target are straight line programs, thus the language is not Turing complete since it contains neither loops nor any recursive structure, but core functionality of safety-critical systems are often implemented in a restricted language similar to the one described in this paper (see *e.g.*, [13,10]).

One of the main challenge is that these systems use real numbers that can not be represented in an exact way in programs and we have to use different representations in order to make computation, *e.g.*, the floating point numbers defined by the IEEE754-Standard [9] with a fixed number of bits. This kind of representations always introduces differences between the *expected* behavior (defined by the abstract semantics), where we assume that numbers are genuine real numbers, and the *actual* behavior (defined by a concrete semantics), happening when we run the program. Furthermore some programs, *e.g.*, air traffic management systems [13], use tests on comparisons between real numbers and we may want to ensure some numerical stability on the programs that use these features since a tiny error in a test can make the actual behavior to greatly diverge from the expected one.

Using formalizations of the floating point semantics [2,11,8,12] makes proofs of programs used in aeronautics difficult since most of the properties of real numbers and operations (*e.g.*, associativity) do not hold on such representations, therefore many proofs are done on the abstract semantics which does not represent the actual behavior of the program. Differences between the abstract and the floating point semantics have been studied in a particular efficient way using static analysis by abstract interpretation with numerical abstract domains [4] and interval arithmetic in order to provide numerical analysis of programs [7]. These methods have been used to define a program transformation to improve accuracy [11] but our goal is slightly different.

We aim at producing, for any program that computes a boolean with real arithmetic using  $+$ ,  $-$ ,  $\times$ ,  $/$ ,  $\sqrt{\phantom{x}}$ , an equivalent program whose concrete semantics is equal to its abstract semantics on genuine real numbers. Exact representation of algebraic numbers has already been efficiently implemented [3] and formally represented in the COQ system [5]. However, in embedded systems we favor programs that can be executed in a fixed size memory, this prevent us from using the polynomial representation of algebraic numbers since computation over it requires the use of dynamic structures.

Exact computation over real numbers can also be done on addition and multiplication by using a fixed point representation with dynamic size since we are able to predict the sizes required by such computations using static analysis, the language we target being free of loops or recursion. Thus, we aim at defining a program transformation that removes square roots and divisions operations in every boolean expressions of the program, *e.g.*, transforms  $\sqrt{x} > y$  into  $y < 0 \vee x > y^2$ , in order use this exact computation with only addition and multiplication which will protect the control flow from rounding errors.

Our goal being to improve the safety of the systems we target, we want to formally prove in PVS that this transformation preserves the semantics. Therefore, by computing exactly with addition and multiplication, we can ensure that the path taken in each test is the same in both abstract and concrete semantics and that formal proofs of properties about boolean values done on the abstract semantics still hold on the concrete one. Indeed, if proofs on the original program using concrete semantics are difficult, so are proofs on the output of program transformations due to the size and the complexity of the produced code, this is the reason why the correction of the transformation needs to be formally proven.

The paper is organized as follows. First, we define the language on which the transformation applies and some general features about the PVS formalization. Then we define the main transformation and two auxiliary methods that respectively remove square roots and divisions from boolean expression and from variable definitions. Finally, we present some experimental results using the OCaml implementation of the transformation which is almost an executable copy of PVS formalization [1] with few hand made extra features.

---

<sup>1</sup> PVS and OCaml files are available at <http://www.lix.polytechnique.fr/~neron/>

# 1 Presentation of the Language

## 1.1 Language Definition

In this section we define the syntax of the language the transformation applies to. This language, that embeds the core functionalities of our targeted programs, is a typed functional language that contains numerical ( $\mathbb{R}$ ) and boolean constants, variable definitions (as `let` in instructions), tests (`if then else`), pairs and the usual arithmetic  $+$ ,  $-$ ,  $\times$  (we also use  $.$  instead of  $\times$ ),  $/$ ,  $\sqrt{\phantom{x}}$ , the comparisons  $=$ ,  $\neq$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$  and boolean operators ( $\wedge$ ,  $\vee$ ,  $\neg$ ). Therefore we can define the syntax of our language as follows:

**Definition 1.1 (Syntax of the language).**

$$\text{Prog} := \begin{array}{l} \text{Const} \quad | \text{fst Prog} \quad | \text{Var} \\ \text{uop Prog} \quad | \text{snd Prog} \quad | \text{let Var} = \text{Prog in Prog} \\ \text{Prog op Prog} \quad | (\text{Prog}, \text{Prog}) \quad | \text{if Prog then Prog else Prog} \end{array}$$

where:

$$\begin{array}{l} \text{Const} \subset \mathbb{R} \cup \{\text{True}, \text{False}\} \\ \text{op} \in \{+, \times, /, =, \neq, >, \geq, <, \leq, \wedge, \vee\} \\ \text{uop} \in \{\sqrt{\phantom{x}}, -, \neg\} \end{array}$$

Then we introduce the type system, as usual we use a typing environment  $\Gamma$  that associates to every free variable its type.

**Definition 1.2 (Type system).**

$$\text{Type} := \mathbb{R} \mid \mathbb{B} \mid \text{Type} \times \text{Type}$$

The rules are the usual ones for a functional language, e.g. [2](#):

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2} \qquad \frac{\Gamma \vdash f : \mathbb{B} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if } f \text{ then } e_1 \text{ else } e_2 : T}$$

These types are used to identify the way a program has to be transformed. Indeed, the transformation is different for pure numerical expressions (e.g., in a variable definition) and for the ones used in boolean expressions (i.e., as arguments of a comparison). It is easy to define a type checking function in PVS, `type_infer(p,  $\Gamma$ )`, that returns either a type or an undefined value ( $\mathbb{U}$ ) if the program has no valid type in the given environment.  $Ty_\Gamma(p)$  now denotes the result of `type_infer( $\Gamma$ )(p)`.

**Notation.** In the paper we use the `teletype font` to represent PVS expressions. For clarity and concision reasons, every element of `Prog` and subtypes of `Prog` will be written using the `sans serif font` in the concrete syntax (instead of their PVS abstract syntax, e.g., we will write `e1 op e2` for `bop(op, e1, e2)`), some PVS expressions will be abstracted by their equivalent *mathematic expressions* (e.g.,  $\forall \Gamma, Ty_\Gamma(p) \neq \mathbb{U}$  stands for `FORALL env, type_infer(p, env) /= Undefined`)

<sup>2</sup> Complete sets of rules and formal definitions can be found in the long version [14](#)

and we will not give the type when there is no ambiguity ( $\Gamma$  is always a typing environment *i.e.*, a function from  $\text{Var}$  to  $\text{Type}$ ).

We then define the denotational semantics of a program in the language, using an environment  $\text{Env}$  that associates to every variable its value. It is the usual semantics of a functional language. The  $\text{Fail}$  value corresponds to the square roots of negative numbers, divisions by zero and unsound types cases.

**Definition 1.3 (Denotational semantics of the language).** *The rules for square root and test are:*

$$\frac{\text{Env} \vdash \llbracket E \rrbracket = e}{\text{Env} \vdash \llbracket \sqrt{E} \rrbracket = \text{Fail}} \quad e < 0 \vee e = \text{Fail} \qquad \frac{\text{Env} \vdash \llbracket E \rrbracket = e}{\text{Env} \vdash \llbracket \sqrt{E} \rrbracket = \sqrt{e}} \quad e \geq 0$$

$$\frac{\text{Env} \vdash \llbracket f \rrbracket = \text{True} \quad \text{Env} \vdash \llbracket E_1 \rrbracket = e_1}{\text{Env} \vdash \llbracket \text{if } f \text{ then } E_1 \text{ else } E_2 \rrbracket = e_1} \qquad \frac{\text{Env} \vdash \llbracket f \rrbracket = \text{Fail}}{\text{Env} \vdash \llbracket \text{if } f \text{ then } E_1 \text{ else } E_2 \rrbracket = \text{Fail}}$$

We now denote  $\llbracket P \rrbracket_{\text{Env}}$  the abstract semantics of  $P$  into an environment  $\text{Env}$  (*i.e.*, the  $v$  such that  $\text{Env} \vdash \llbracket P \rrbracket_{\text{Env}} = v$ , that corresponds to the PVS function **semantics**). The language being defined, we now precise some subtypes of  $\text{Prog}$ , that represent restricted syntactic forms.

### 1.2 Program Sub-types

**Normalized Language.** The normalized language is the subtype on which our transformation applies, it is a subtype of the  $\text{Prog}$  type that have the following definition (see [\[4\]](#) for program normalization):

**Definition 1.4 (Expressions and programs normal form).** *The unary expressions  $E_u$  are built with operators, the expressions  $E$  with pairs and the programs  $P$  allow variable definitions and tests.*

$$\begin{aligned} E_u &:= \text{Var} \mid \text{Const} \mid \text{uop } E_u \mid E_u \text{ op } E_u \mid \text{fst } E_u \mid \text{snd } E_u \\ E &:= (E, E) \mid E_u \\ P &:= \text{let } \text{Var} = P \text{ in } P \mid \text{if } P \text{ then } P \text{ else } P \mid E \end{aligned}$$

Hence, in a well typed program, projections can not contain any square roots or divisions as sub expressions.

**Target Language.** Now we present the language that corresponds to programs from which divisions and square roots have been eliminated. Certainly, we can not eliminate all square roots and divisions from any program, (*e.g.*, in the program  $\sqrt{2}$ ) but we are able to remove them from all the boolean values computations. Therefore we define some new subtypes of expressions and programs.

**Definition 1.5 (E and P sub-classes)**

- $E_N$  is  $E$  where the  $\sqrt{\quad}$  and  $/$  operators are not allowed
- $E_{N_{\sqrt{\cdot}/\cdot}}$  is  $E$  where the  $\sqrt{\quad}$  and  $/$  operators are allowed only in numerical expressions that are not arguments of comparison operators
- $P_N := \text{let } \text{Variable} = P_N \text{ in } P_N \mid \text{if } P_N \text{ then } P_N \text{ else } P_N \mid E_N$  is the class of program that do not contain any  $\sqrt{\quad}$  or  $/$
- $P_{N_{\sqrt{\cdot}/\cdot}} := \text{let } \text{Variable} = P_N \text{ in } P_{N_{\sqrt{\cdot}/\cdot}} \mid \text{if } P_N \text{ then } P_{N_{\sqrt{\cdot}/\cdot}} \text{ else } P_{N_{\sqrt{\cdot}/\cdot}} \mid E_{N_{\sqrt{\cdot}/\cdot}}$  is  $P$  without  $\sqrt{\quad}$  or  $/$  in tests or variable definitions bodies

For example  $(\sqrt{x}, a > b)$  is in  $E_{N_{\sqrt{\cdot}}}$  but not in  $E_N$  but  $\sqrt{a} > b$  is in none of them. One can notice that any program in  $P_{N_{\sqrt{\cdot}}}$  of type  $\mathbb{B}^n$  is also in  $P_N$ . These definitions allow us to characterize the set of programs transformed by each step of our transformation and what kind of programs it produces.

**PVS Sub-typing.** The specification of an algorithm and the related proofs in PVS mainly relies on the PVS sub-typing. Given a type  $T$  and a predicate  $P$  of type  $T \rightarrow \mathbb{B}$ ,  $\{x : T \mid P(x)\}$  is the subtype of  $T$  of all elements  $x$  of type  $T$  that verify  $P$ , this type can also be denoted  $(P)$ . Then every definition of a function in PVS can be specified using these subtypes, *e.g.*,

$$f(x : (P)) : \{x' : T' \mid P'(x')\}$$

defines a partial function on  $T$  that takes only elements  $x$  of type  $T$  that verify  $P$  and returns elements of type  $T'$  that verify  $P'$ . Then PVS type checker generates Type Check Conditions (TCC) where we have to prove that:

- $f$  can be applied to every element of type  $(P)$  (*Completeness*)
- $\forall x : T, f(x) : T' \wedge P'(f(x))$  (*Soundness*)
- if  $f$  is recursive, then for every recursive call on  $e$ :
  - $e : (P)$  (*Recursive call correctness*)
  - a measure decreases, according to a well founded order provided in the definition of  $f$  (*Termination*)

The type of a function can also be restricted using the `HAS_TYPE` judgement, *e.g.*, given two types  $T$  and  $T'$ , two subtypes  $S$  of  $T$  and  $S'$  of  $T'$  and a function  $f(x : T) : T'$ , then we can state the following judgement  $f(x : S) \text{ HAS\_TYPE } S'$  and use either  $T'$  or  $S'$  as type of  $f(x)$  when  $x : S$ . Therefore we will not give in this paper correctness lemmas of the functions we define but only their type.

### 1.3 No Fail Assumption

In the context of that transformation, we assume that the programs we want to transform are well typed (there exists a type environment that allow us to type the program) and do not fail in the environment where we want to evaluate them (*e.g.*, there are no divisions by zero or square roots of negative numbers) since it is the case of embedded programs we target. Hence we will not have to force the failure cases that disappear when removing divisions and square roots, *e.g.*, we can transform  $1/x > 0$  into  $x > 0$  instead of if  $x = 0$  then Fail else  $x > 0$ . Therefore we will only ensure the preservation of the type and the semantics in every environment where the initial program is well typed and does not fail. This preservation is defined in the language of PVS by the predicate `sem_ty_nf_eq(p)(p')`, that is:

$$\begin{aligned} (\forall \Gamma, Ty_{\Gamma}(p) \neq \mathbb{U} \implies Ty_{\Gamma}(p') = Ty_{\Gamma}(p)) \wedge \\ (\forall Env, \llbracket p \rrbracket_{Env} \neq Fail \implies \llbracket p' \rrbracket_{Env} = \llbracket p \rrbracket_{Env}) \end{aligned}$$

We consider that programs  $p$  and  $p'$  are *equivalent* when  $\text{sem\_ty\_nf\_eq}(p)(p')$  holds. Given this no fail hypothesis, we can now formally define our main goal which is to present a transformation  $\text{Elim}$  that has the following specification:

**Definition 1.6 (Elim specification)**  
 $\text{Elim}(\Gamma)(p : \text{Prog} \mid \text{Ty}_\Gamma(p) \neq \mathbb{U}) : \{p' : \text{P}_{N_{\check{v},/}} \mid \text{sem\_ty\_nf\_eq}(p)(p')\}$

The  $\text{Elim}$  function is defined by a sequence of elementary transformations, all of them preserving the semantics when the program does not fail. Every elementary transformation is formalized using a function whose output type is of the form:

$$\{p' : P' \mid \text{sem\_ty\_nf\_eq}(p)(p')\} \quad \text{where } P' \text{ is a subtype of Prog}$$

### 1.4 Generic Functions

**Substitution.** We now describe a function that defines a capture avoiding substitution of every free occurrence of a variable  $x$  in a program  $p$  by a given program  $e$  (also denoted  $p[x:=e]$ ). The semantics and the types of  $p$  are preserved in every environment where  $e$  is well typed and does not fail. That gives the following specification:  $\text{replace}(x : \text{Var}, e, p : \text{Prog}) : \{p' : \text{Prog} \mid \forall \Gamma, Env,$

$$\llbracket p' \rrbracket_{Env} = \llbracket p \rrbracket_{Env;(x, \llbracket e \rrbracket_{Env})} \wedge \text{Ty}_\Gamma(p') = \text{Ty}_\Gamma(x, \text{Ty}_\Gamma(e))(p)\}$$

Termination is ensured using the number of variable definitions. The  $\text{replace}$  function also has several types that ensure the preservation of subtypes, *e.g.*,

$$\text{replace}(x : \text{Var}, e : E_u, p : P) \text{ HAS\_TYPE } P$$

Correction of the substitution is proved using the lemmas that characterize the fresh variables, *e.g.*, if we denote  $FV(p)$  the set of the free variables of  $p$ , then:

$$\begin{aligned} & - \text{no\_FV\_no\_change} : \text{LEMMA} = \\ & \quad \forall x, p, x \notin FV(p) \implies \forall Env, v, \llbracket p \rrbracket_{Env} = \llbracket p \rrbracket_{Env;(x,v)} \end{aligned}$$

**Program Normalization.** The transformation of any program in  $\text{Prog}$  into a program of type  $P$  can be done using a set of reduction rules that takes the tests and variable definitions out of the binary operators and pairs and that reduces the projections, *e.g.*:

- $\text{uop}(\text{let } x = e1 \text{ in } e2) \longrightarrow \text{let } x = e1 \text{ in } (\text{uop } e2)$
- $(\text{if } f \text{ then } e1 \text{ else } e2) \text{ op } e3 \longrightarrow \text{let } xi = \text{if } f \text{ then } e1 \text{ else } e2 \text{ in } (xi \text{ op } e3) \quad xi \notin FV(e3)$
- $\text{fst}(e1, e2) \longrightarrow e1$

The transformation rules for a test inside a binary operator or a pair uses a variable definition in order to avoid duplicating the other arguments, a process that can strongly increase the size of the transformed code. On the other way, creating new variable definition makes the direct termination proof more complicated. Therefore we first define a set of functions that normalize the application of operators, pair or projections to a program in normal form, *e.g.*, :

$$\text{uop\_P\_switch}(\text{uop})(p:P \mid \text{wtp}(p)) : \{p':P \mid \text{sem\_ty\_nf\_eq}(\text{uop } p)(p')\}$$

$\text{wtp}(p)$  being the predicate stating that the program  $p$  is well typed. Since we sometimes use renaming, due to free variable constraints, in the case of binary operators, the termination proof is done using the depth of the program. The correction is proved by using the `no_FV_no_change` lemmas. Using these definitions we can define the main function that transforms any program into an equivalent normalized program by iterating these functions:

$$\text{p\_norm\_reduction}(\text{xi}:\text{Var})(p:(\text{wtp})) : \{pp:P \mid \text{sem\_ty\_nf\_eq}(p)(pp)\}$$

Since we can transform every well typed program in `Prog` into an equivalent one in `P`, we now describe the major function `Elim` of the transformation that transforms any well typed program in `P` into an equivalent one in  $\mathbb{P}_{\mathbb{N}_{\sqrt{\cdot}}}$ .

## 2 Normalized Program Transformation

This `Elim` function is a recursive program transformation algorithm eliminating square roots and divisions. It uses two functions, `Elim_bool` and `Elim_let`, that will be described in sections 3 and 4 and have the following specifications:

**Definition 2.1** (*Elim\_bool and Elim\_let specifications*). *Elim\_bool transforms any comparison (the comparison\_expression subtype will be formally defined in section 3) into an equivalent boolean program without any square root or division and Elim\_let transforms a variable definition into an equivalent one that contains neither square root nor division in its body:*

$$\text{Elim\_bool}(\Gamma)(\text{xsq} : \text{Var})(c : \text{comparison\_expression}(\Gamma)) : \\ \{ e : \mathbb{B}_{\text{let}} \mid \text{sem\_ty\_nf\_eq}(c)(e) \}$$

where  $\mathbb{B}_{\text{let}}$  is the type of square root and division free boolean expressions that allows variable definitions and `xsq` a variable used to name some boolean expressions in order to avoid formula duplications.

$$\text{Elim\_let}(\Gamma)(x:\text{Var}, p1:\mathbb{P}_{\mathbb{N}_{\sqrt{\cdot}}}, p2:P \mid \text{Ty}_\Gamma(\text{let } x = p1 \text{ in } p2) \neq \mathbb{U}) : \\ \{ x':\text{Var}, p1':\mathbb{P}_{\mathbb{N}}, p2':P \mid \\ \text{sem\_ty\_nf\_eq}(\text{let } x = p1 \text{ in } p2)(\text{let } x' = p1' \text{ in } p2') \ \& \\ \text{if\_letin\_number}(p2') \leq \text{if\_letin\_number}(p2) \}$$

*Remark.* By applying the `Elim_bool` function to every comparison we can find in an expression, we can define the `Elim_bool_expr` function that transforms every well typed expression in `E` into an equivalent one in  $\mathbb{E}_{\mathbb{N}_{\sqrt{\cdot}}}$ .

The function `if_letin_number(p)` gives the number of variable definitions and tests which occur in  $p$ , it will be used to prove the termination of the main algorithm. Therefore using these two functions, we can define the recursive algorithm that transforms any program in `P` in an equivalent one in  $\mathbb{P}_{\mathbb{N}_{\sqrt{\cdot}}}$ :

**Definition 2.2 (Elim function)**

```

Elim( $\Gamma$ , xsq)(p:P |  $Ty_{\Gamma}(p) \neq Fail$ ) : {p': $P_{N_{\sqrt{\cdot}/\cdot}}$  | sem_ty_nf_eq(p)(p')}=
  CASES p OF
    let x = p1 in p2 :
      LET pn1 = Elim( $\Gamma$ )(xsq)(p1) IN
      LET (x',p1',p2') = Elim_let( $\Gamma$ )(x,pn1,p2) IN
      let x' = p1' in Elim( $\Gamma$ , (x',  $Ty_{\Gamma}(p1')$ ))(xsq)(p2'),
    if f then p1 else p2 :
      if Elim( $\Gamma$ )(xsq)(f) then Elim( $\Gamma$ )(xsq)(p1) else Elim( $\Gamma$ )(xsq)(p2)
  ELSE Elim_bool_expr( $\Gamma$ )(xsq)(p) ENDCASES
  MEASURE (if_letin_number(p)) BY <

```

The correction is ensured by both `Elim_bool_expr` and `Elim_let` typing predicates. Therefore, applying the `main_elim` function, which is the composition of both `p_norm_reduction` and `Elim` functions to any program in `Prog`, we can state that every program in `Prog` has an equivalent one in  $P_{N_{\sqrt{\cdot}/\cdot}}$ :

**Theorem (Prog is equivalent to  $P_{N_{\sqrt{\cdot}/\cdot}}$ )**

$\forall \Gamma, xsq, ifname,$   
 $main\_elim(\Gamma)(xsq,ifname)(p : Prog |  $Ty_{\Gamma}(p) \neq Fail$ ) :$   
 $\{p':P_{N_{\sqrt{\cdot}/\cdot}} | sem\_ty\_nf\_eq(p)(p')\}$

And its corollary stating that any boolean program (whose type is in  $\mathbb{B}^n$ ) has an equivalent division and square root division free program:

**Corollary (Square root and division elimination in boolean programs)**

$\forall \Gamma, xsq, ifname,$   
 $main\_elim(\Gamma)(xsq,ifname)(p : Prog |  $Ty_{\Gamma}(p) \in \mathbb{B}^n$ ) HAS\_TYPE  $P_N$$

Let us focus on the `Elim_bool` function in section 3 and on the `Elim_let` function in section 4

### 3 $\sqrt{\cdot}$ and / Elimination in Boolean Expressions: `Elim_bool`

In this section, we describe the function `Elim_bool`, *i.e.*, how to transform a well typed comparison expression into an equivalent boolean expression which is free of divisions and square roots (*i.e.*, of class  $B_{let}$ ). The elimination of square roots and divisions in boolean formulas is a particular case of the quantifier elimination over real closed fields *e.g.*, the formula  $\sqrt{x + \frac{3}{\sqrt{y}}} > 4$  can be rewritten as:

$$\exists x', y', y'', x'^2 = x \wedge y'^2 = y \wedge y'' \times y' = 3 \wedge x' + y'' > 4$$

Therefore, quantifier elimination produces an equivalent boolean formula without any division or square root. The general quantifier elimination with cylindrical algebraic decomposition [15][6] has been implemented as `QEPCAD` [3] but this algorithm does not succeed in most of our cases due to the large number of

<sup>3</sup> See <http://www.usna.edu/cs/~qepcad/B/QEPCAD.html>



free variables in a program transformation. The elimination of square roots and division in formulas has also been studied by V. Weispfenning in its restriction of the quantifier elimination to the quadratic case [16] but we use a different one that allows us to eliminate all occurrences of one square root in one step.

### 3.1 Expressions Subtyping

As we mentioned in section 1, some sub-types of `Prog` will help us to define some specific transformations. `Elim_bool` is a partial function that only applies to a sub-type of the general programs that is:

$$\text{comparison\_expression}(\Gamma) : \text{TYPE} = \{ e1 \text{ op } e2 : \text{Prog} \mid \\ \text{Ty}_\Gamma(e1 \text{ op } e2) = \mathbb{B} \ \& \ \text{op} \in \{=, \neq, >, \geq, <, \leq\} \ \& \ e1 : \mathbb{N}_{\sqrt{, /}} \ \& \ e2 : \mathbb{N}_{\sqrt{, /}} \}$$

where  $\mathbb{N}_{\sqrt{, /}}$  is the sub-type of numerical expressions ( $\mathbb{E}_u$  with only numerical operators). We also define the type of well typed numerical expressions:

$$\text{wt\_num\_expr}(\Gamma) : \text{TYPE} = \{ e : \mathbb{N}_{\sqrt{, /}} \mid \text{Ty}_\Gamma(e) = \mathbb{R} \}.$$

Using these type definitions we can define the first step of elimination of square roots and divisions which is a reduction to a normal form for numerical expressions  $\mathbb{N}_{\sqrt{, /}}$ .

### 3.2 One Division Normal Form

The elimination of the division is in fact quite simple since every numerical expression on  $+$ ,  $-$ ,  $\times$ ,  $/$ ,  $\sqrt{\quad}$  can be represented with only one division at head. Therefore the expression we get by reducing expressions to the same denominator without splitting square roots corresponds to the following normal form:

**Definition 3.1 (Division and polynomial normal forms).**

$$\text{DNF} := \text{PNF} \mid \frac{\text{PNF}}{\text{PNF}} \\ \text{PNF} := \text{PNF} + \text{PNF} \mid \text{PNF} \times \text{PNF} \mid - \text{PNF} \mid \sqrt{\mathbb{N}_{\sqrt{, /}}} \mid \text{fst PNF} \mid \text{snd PNF} \mid \text{Const} \mid \text{Var}$$

Reduction to division normal form is done by the `to_dnf` function that transforms any well typed numerical expression into an equivalent one with only one division as head operation. This corresponds to the following specification:

$$\text{to\_dnf}(\Gamma)(e : \text{wt\_num\_expr}(\Gamma)) : \\ \{ \text{eout} : \text{DNF} \mid \text{sem\_ty\_nf\_eq}(e)(\text{eout}) \ \& \ \text{sq\_number\_eq}(e, \text{eout}) \}$$

that implements a reduction to the same denominator.

*Remark.* This transformation is only correct in the context of the no fail assumption, e.g., semantics equivalence of the rule  $\frac{e1}{e2} \longrightarrow \frac{e1.e3}{e2}$  only holds when  $\llbracket e3 \rrbracket_{Env} \neq 0$ .

`sq_number_eq(e, eout)` is a predicate that will be used to prove the termination of the `Elim_bool` function. It states that every square root that appears in the output was already in the input:

$$\text{sq\_number\_eq}(e, \text{eout}) = \forall \text{sq}, \sqrt{\text{sq}} \leq \text{eout} \Rightarrow \sqrt{\text{sq}} \leq e$$

where  $x \leq y$  means that  $x$  is a subterm of  $y$  or is equal to  $y$ .

### 3.3 Division Elimination Rules

Once we have a comparison between two DNF form, the head division can be easily eliminated by multiplying both arguments by the denominators. But since we want to avoid the creation of new comparisons due to the signs conditions of the denominators when multiplying, we will sometimes prefer the following elimination rule:

**Definition 3.2 (Elimination of division)**

- when  $\mathfrak{R} \in \{=, \neq\}$ :  $\frac{e1}{e2} \mathfrak{R} \frac{e3}{e4} \longrightarrow e1.e4 - e3.e2 \mathfrak{R} 0$
- when  $\mathfrak{R} \in \{>, <, \geq, \leq\}$ :  $\frac{e1}{e2} \mathfrak{R} \frac{e3}{e4} \longrightarrow e1.e2.e4^2 - e3.e4.e2^2 \mathfrak{R} 0$

*Remark.* The application of this rule eliminates the last division that is not inside a square root. Therefore the expressions produced are boolean combinations of relations between numerical expressions in PNF and 0.

These elimination rules are implemented in PVS with the following function:

```
elim_div_rule1( $\Gamma$ )(p : comparison_expression( $\Gamma$ )) :
  { e1 op e2 : comparison_expression( $\Gamma$ ) | sq_number_eq(p,e1) &
    sem_ty_nf_eq(p)(e1 op e2) & PNF?(e1) & e2 = 0 }
```

The OCaml implementation gives the option to use either this rule or the usual one (with sign distinction) but only this one was proven since it gives better results for the size of the produced code. Now we first have to eliminate top level square roots, that will make the divisions in the arguments of square roots appear at top level and allow us to continue the elimination of both operations.

### 3.4 Square Root Elimination

For every comparison between a PNF form and 0 we choose one square root that is at top-level, that means this square root does not appear in the argument of another square root, we call it Q. Then, we factorize the PNF expression in the form  $(P.\sqrt{Q} + R)$  where  $\sqrt{Q}$  does not appear in P or R:

```
factorize_sqrt( $\Gamma$ )(e:wt_num_expr( $\Gamma$ ) | PNF?(e))(q:wt_num_expr( $\Gamma$ )) :
  { p,r : wt_num_expr( $\Gamma$ ) | sem_ty_nf_eq(e)(p.\sqrt{q} + r)
    sq_number_but_q_eq(e)(q) & sq_number_but_q_eq(e)(q)(r) }
```

where `sq_number_but_q_eq` states that every square root of an output was already in an input and is different from q. Then by splitting cases depending on the signs of P and R we get a new formula equivalent to  $P.\sqrt{Q} + R > 0$  under the condition that  $Q \geq 0$ :

$$(P.\sqrt{Q} + R) > 0 \longrightarrow (P > 0 \wedge R > 0) \vee (P > 0 \wedge P^2.Q - R^2 > 0) \vee (R > 0 \wedge R^2 - P^2.Q > 0)$$

In order to reduce the size of the final expression, we name the atoms and share their different occurrences instead of duplicating them. Therefore the implementation of these rules relies on composition of functions such as

```

name_comp(xsq : Var, e1, e2, e3, e4 : Prog) : Prog =
  let xsq = ((e1,e2),(e3,e4)) in
    (fst(fst(xsq)) ∧ snd(fst(xsq))) ∨ (fst(fst(xsq)) ∧ fst(snd(xsq))) ∨
      (snd(fst(xsq)) ∧ ¬ fst(snd(xsq)) ∧ snd(snd(xsq)))

```

and on elimination rules such as

```

elim_sqrt_rule_gt(Γ)(p,q,r : wt_num_expr(Γ)) :
  { e1, e2, e3, e4 : comparison_expression(Γ) |
    ∀(x:Var) : sem_ty_nf_eq(p.√q + r > 0)(name_comp(x,e1,e2,e3,e4)) } =
    (p > 0, r > 0, p2.q - r2 > 0, p2.q - r2 ≠ 0)

```

Using reduction to DNF, square root factorization and the elimination rules for division and square root, we can define the main algorithm which transforms a comparison into an equivalent fragment of code that contains neither division nor square root. This algorithm is a recursive combination of these transformations:

**Definition 3.3 (elim\_bool description).** *While the comparisons contains divisions or square roots, do:*

- Reduce to DNF
- Eliminate the head division
- Factorize using one top level square root
- Eliminate that square root

The full transformation of the comparisons is defined by the `elim_bool` function whose specification was given in definition 2.1. In order to ensure the termination of the algorithm, we prove that the number of distinct square roots a comparison contains strictly decreases. Indeed, after applying the 4 steps, every square root in the output comparisons was already in the input and is different from the one we eliminated, which was also in the input.

### Example 3.1 (/ and √ elimination)

```

 $\frac{x+\sqrt{y}}{z} > 0 \longrightarrow$ 
  let xsq = ((z > 0, x.z > 0), (z2.y - (x.z)2 > 0, z2.y - (x.z)2 ≠ 0)) in
    (fst(fst(xsq)) ∧ snd(fst(xsq))) ∨ (fst(fst(xsq)) ∧ fst(snd(xsq))) ∨
      (snd(fst(xsq)) ∧ ¬ fst(snd(xsq)) ∧ snd(snd(xsq)))

```

*Complexity.* In a comparison  $A$ , if we write  $|A|_{\sqrt{\cdot}}$  the number of distinct square roots, the number of comparisons produced by eliminating the square roots in this comparison is bounded by  $4^{|A|_{\sqrt{\cdot}}}$ . We did not study the exact complexity of the algorithm since the reduction into DNF and the factorization may also increase the size of the formula. Nevertheless, experimentally our specialized algorithm is able to transform bigger formula than the QEPCAD implementation of the quantifier elimination by cylindrical algebraic decomposition, *e.g.*,

$$\exists sq, sq^2 = q \wedge sq \geq 0 \wedge (p_1.sq + r_1).(p_2.sq + r_2).(p_3.sq + r_3) > 0$$

Indeed, the complexity of our transformation does not depend on the number of variables of the formula but only on the number of square roots and divisions.

We now define the second transformation used in the method, it transforms a variable definition into an equivalent one without square roots or divisions.

## 4 Transformation of Variables Definitions: Elim\_let

In this section, our goal is to define how to transform a variable definition in order to take the divisions and square roots out of the body. We will describe a function that corresponds to the specification given in definition 2.1. We noticed that inlining the variable definitions satisfies this specification but it leads to an explosion of the size of the code therefore we will now present a way to inline only the divisions and square roots as in example 4.1.

There are different way to achieve such a goal and we really want to minimize the number of square roots created in that process, therefore for genericity reason this function is only axiomatized in Pvs but the OCaml program implements one version. Hence, in this section, we only use the *mathematic* font in order to describe the transformation and for clarity we use multiple variable definition.

### Example 4.1

$$\begin{aligned} \text{let } x = \text{if } y > 0 \text{ then } (a1 + a2)/b \text{ else } c + \sqrt{d1.d2} \text{ in } P &\longrightarrow \\ \text{let } (x0,x1,x2) = \text{if } y > 0 \text{ then } (a1 + a2,b,0) \text{ else } (c,1,d1.d2) \text{ in } P[x:= \frac{x0+\sqrt{x2}}{x1}] \end{aligned}$$

Square roots and divisions that were used to define  $x$  are now explicit in  $P$ .

### 4.1 Transformation of the Variable Definition Code

The elimination of square roots and divisions from a variable definition (let  $x = p1$  in  $p2$  where  $x : \text{Var}$ ,  $p1 : P_{N_{\sqrt{\cdot}}}$  and  $p2 : P$ ) relies on a decomposition of its body  $p1$  in:

- a *program part*  $Pp$  of type  $E_{N_{\sqrt{\cdot}}}^n \longrightarrow P_{N_{\sqrt{\cdot}}}$  and  $E_N^n \longrightarrow P_N$
- a list of expressions  $(e_1, \dots, e_n)$  of  $E_{N_{\sqrt{\cdot}}}^n$  such that  $Pp(e_1, \dots, e_n) = p1$ , that decomposes itself in:
  - a *template*  $Temp$  of type  $E_N^k \longrightarrow E_{N_{\sqrt{\cdot}}}$
  - a list of  $k$ -tuple of division and square root free *sub-expressions*  $se_1, \dots, se_n$  of type  $E_N^k$  such that:  $\forall i \in [1\dots n], Env, \llbracket Temp(se_i) \rrbracket_{Env} = \llbracket e_i \rrbracket_{Env}$

Therefore we have that  $\forall Env, \llbracket p1 \rrbracket = \llbracket Pp(Temp(se_1), \dots, Temp(se_n)) \rrbracket_{Env}$  and we transform the variable definition in the following way:

**Definition 4.1 (Variable definition transformation).** *A variable definition is transformed by commuting elements of its decomposition:*

$$\begin{aligned} \text{let } x = Pp(Temp(se_1), \dots, Temp(se_n)) &\longrightarrow \text{let } (x_{\epsilon_1}, \dots, x_{\epsilon_k}) = Pp(se_1, \dots, se_n) \\ \text{in } p2 &\qquad \qquad \qquad \text{in } p2[x := Temp(x_{\epsilon_1}, \dots, x_{\epsilon_n})] \end{aligned}$$

We now begin the description of the two decompositions. In order to simplify the presentation, we introduce the following notation. For every program  $p$ ,  $(fun(x_1, \dots, x_n) \rightarrow p)$  is a function of  $P^n \longrightarrow P$ , such that:

$$(fun(x_1, \dots, x_n) \rightarrow p)(u_1, \dots, u_n) = p[x_1 := u_1; \dots; x_n := u_n]$$

We first decompose the body of the definition that is in  $P_{N_{\sqrt{\cdot}}}$  into a *program part* (the part that contains the local variable definitions and tests) and the *expression part*.

## 4.2 Program and Expression Part Decomposition

We define the following recursive algorithm *Decompose*, that computes from a program  $p$  in  $P_{N_{\sqrt{\cdot},/}}$ , its *program part* and its *expression part*.

### Definition 4.2 (Program and expression part decomposition).

- $Decompose(p) =$
- if  $p \in E_{N_{\sqrt{\cdot},/}}$  then  $((fun\ x \rightarrow x), [p])$
  - if  $p = let\ y = a\ in\ p'$  then
    - $(pp, ep) := Decompose(p')$
    - return  $((fun\ x \rightarrow let\ y = a\ in\ pp(x)), ep)$
  - if  $p = if\ B\ then\ p_1\ else\ p_2$  then
    - $(pp_1, ep_1) := Decompose(p_1)$
    - $(pp_2, ep_2) := Decompose(p_2)$
    - return  $((fun\ (x_1, x_2) \rightarrow if\ B\ then\ pp_1(x_1)\ else\ pp_2(x_2)), ep_1 @ ep_2)$

### Example 4.1

$$Decompose(\text{if } F \text{ then let } y = a \text{ in } a + \sqrt{b} \text{ else } c) = \\ (fun\ (x, y) \rightarrow \text{if } F \text{ then let } y = a \text{ in } x \text{ else } y, (a + \sqrt{b}, c))$$

The program  $p$  being in  $P_{N_{\sqrt{\cdot},/}}$ , neither the local variable definitions bodies nor the boolean arguments of the tests can contain division or square root. Therefore if we apply  $Pp$  to a tuple of expressions in  $E_N$ , the result does not contain any divisions or square roots. Now we will see how we can decompose the expression part in order to remove square roots and divisions from it.

## 4.3 Expression Decompositions

Initially we will see how to decompose a single expression by assuming that the body of the definition does not contain any test, thus the expression part is a list of one element, then we will extend this definition to any list of expressions.

**Template of an Expression.** The idea for transforming a variable definition with an expression is to transform this definition into several variable definitions that correspond to the sub-expressions that are square roots and divisions free. Therefore we need to introduce this decomposition of an expression as the application of a function to square roots and divisions free expressions:

**Definition 4.3 (Template of expression).** *Given an expression  $e \in E_{N_{\sqrt{\cdot},/}}$ , a template for  $e$  is a function  $t$  of  $E_N^k \rightarrow E_{N_{\sqrt{\cdot},/}}$  such that:*

$$\exists e_1, \dots, e_k : E_N, \quad e = t(e_1, \dots, e_k)$$

The template and sub-expression tuple of a unique expression can be computed by giving the tuple of square roots and division free sub-expressions.

**Example 4.2**

$$\frac{a_1 \cdot a_2 + b\sqrt{c}}{d \cdot \sqrt{e_1 + e_2} + \sqrt{c}} = (fun (x_1, x_2, x_3, sq_1, sq_{21}) \rightarrow \frac{x_1 + x_2 \sqrt{sq_1}}{x_3 \sqrt{sq_{21}} + \sqrt{sq_1}})(a_1 \cdot a_2, b, d, c, e_1 + e_2)$$

We use special names for square roots because we want to avoid creating different names for the same square root. Spotting the identical square roots is essential in order to preserve a reasonable size for the produced code.

**Common Template of Expressions.** Given the definition of a template of one expression, we will now extend this decomposition to programs that contain tests. If the body contains a test as in example 4.1, its *expression part* is a list of expressions (*i.e.*,  $[(a_1 + a_2)/b; c + \sqrt{d_1 \cdot d_2}]$ ). The objective is now to decompose these expressions using the same template (*i.e.*,  $\frac{x_0 + \sqrt{x_2}}{x_1}$ ), the common template:

**Definition 4.4 (Common template of expressions).** A common template of two expressions  $e_1$  and  $e_2$  is a function  $t$  that is a template of both  $e_1$  and  $e_2$ , that means  $t$  satisfies:

$$\exists se_1, se_2 : E_{\mathbb{N}}^k, e_1 = t(e_1) \text{ and } e_2 = t(e_2)$$

Let  $x_1, \dots, x_k$  be the local variables corresponding to  $e_1$  and  $x'_1, \dots, x'_{k'}$  the one to  $e_2$ , then if we define  $t$  as:

$$fun (s, x_1, \dots, x_k, x'_1, \dots, x'_{k'}) \rightarrow (s \times e_1 + (1 - s) \times e_2)$$

we have:  $t(1, x_1, \dots, x_k, 0, \dots, 0) = e_1 \wedge t(0, 0, \dots, 0, x'_1, \dots, x'_{k'}) = e_2$ , it is a common template of  $e_1$  and  $e_2$ . This is only an example since we use a more compact form in our transformation in order to limit the number of square roots produced by the template but we will not describe the common template computation in this paper. For example we have the following transformation:

**Example 4.3 (Declaration with test)**

$$\begin{array}{l} \text{let } x = \\ \text{if } F \text{ then } a_1 + a_2 \cdot \sqrt{b_1 + b_2 \cdot \sqrt{b_3}} \\ \text{else } \frac{c_1}{c_2 + c_3 \cdot \sqrt{d_1}} \\ \text{in } P \end{array} \quad \rightarrow \quad \begin{array}{l} \text{let } (x_1, x_2, x_3, x_4, sq_1, sq_2, sq_3) = \\ \text{if } F \text{ then } (a_1, a_2, 1, 0, b_1, b_2, b_3) \\ \text{else } (c_1, 0, c_2, c_3, 0, 0, d_1) \\ \text{in } P[x := t(x_1, x_2, x_3, x_4, sq_1, sq_2, sq_3)] \end{array}$$

where  $t = fun (x_1, x_2, x_3, x_4, sq_1, sq_2, sq_3) \rightarrow \frac{x_1 + x_2 \cdot \sqrt{sq_1 + sq_2 \cdot \sqrt{sq_3}}}{x_3 + x_4 \cdot \sqrt{sq_3}}$  is the common template of  $a_1 + a_2 \cdot \sqrt{b_1 + b_2 \cdot \sqrt{b_3}}$  and  $\frac{c_1}{c_2 + c_3 \cdot \sqrt{d_1}}$ .

The definition of a common template of two expressions extends naturally to several expressions. Therefore we are able to build a common template for any list of expressions in  $E_{\mathbb{N}}^{\sqrt{\cdot}/\cdot}$ .

The transformation being described, the last section presents some remarks about the effective implementation of the transformation.

## 5 Practical Aspects of the Program Transformation

**Exact Computation with  $+$ ,  $-$ ,  $\times$ .** The air traffic management system introduced in [13] computes with distances and other geometric quantities. Therefore

it has use of the square root and division operations. Computing with addition, multiplication and subtraction can be done exactly by using a dynamic representation of real numbers which will allow us to use all the bits we need to avoid losing accuracy during computation (*e.g.*, the product of two numbers of size  $n$  can be stored in a number of size  $2.n$ ) whereas square root or division computations are approximate regardless the number of bits used to express the number. Certainly, these kind of computation does not respect the constraint of embedded systems that requires to know at compile time the memory the program will use at run time. But, since our language does not contain loop or recursion, a simple static analysis can give us the number of bits required by every computation depending the number of bits of the inputs. Being able to compute exactly with addition, multiplication and subtraction is the reason why we want to eliminate square root and division operations.

**Experiments.** We have tested this transformation on several functions defined by the NASA Langley Formal Method Team in their Airborne Coordinated Conflict Resolution and Detection (ACCoRD) framework. Since our language does not support the function definitions yet, we had to inline the calls by hand in the target program. We tried our transformation on the `cd3d` and `trackline` functions defined in [10,13] that solve plane geometry problems. On these two functions we computed the transformation and therefore we can give the size of the produced code but we also computed an estimation of the required memory size if we want to compute on that produced code using exact computation with  $+$ ,  $-$ ,  $\times$ . Assuming the inputs are represented on 64 bits, that gives the following results:

Function	Input code	Output code	Required memory
<code>cd3d</code>	2,3 KB	13 KB	15 Kb
<code>trackline</code>	1 KB	13 KB	57 Kb

The size and the memory required by the output programs quickly growing with the number of square roots and division, it is absolutely critical to keep that number as low as possible during the transformation process in order to get reasonable sizes.

## Conclusion

In this paper is described a way to transform any program built with variable definitions and tests into another semantically equivalent one where the control flow never depends on the result of a square root or division, thus protecting this control flow from rounding errors in these operations. It allows proofs done on the abstract semantic to still hold on the concrete one. This transformation also respects the constraints of embedded code since the transformed program does not use any dynamic structures.

The main issues of this transformation were not only to define procedures that remove divisions and square roots from nearly every part of the program

but also to keep the size of the produced code in an acceptable range. This is the reason why we did not define the computation of the template, since several complex algorithms, as in the OCaml implementation, may be used to generate the most appropriate template in order to minimize the size of the produced code. Nevertheless, the only thing to prove is that they generate a valid template. This work led us to define the problem of template generation that may be of general interest and will be discussed in future work.

Most of the program generic transformations so as the elimination of square root and divisions expressions have already been formalized and proved in the PVS proof assistant. Both the PVS development and the OCaml implementation can be found on the web site of the author. Future work includes extending this transformation to more complex languages which contain structures such as function definitions and bounded loops and keep reducing the size of the produced code by using static analysis techniques such as using the information given by the tests values in the corresponding branches during the transformation.

**Acknowledgement.** I would like to thank both my PhD. advisors Gilles Dowek and Cesar Muñoz for this topic proposal and the helpful discussions and Catherine Dubois and Cyril Cohen for their useful reviews of this paper.

## References

1. Boldo, S., Filliâtre, J.-C.: Formal verification of floating-point programs. In: Kornerup, P., Muller, J.-M. (eds.) Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France, pp. 187–194 (June 2007)
2. Boldo, S., Muñoz, C.: A formalization of floating-point numbers in PVS. Report NIA Report No. 2006-01, NASA/CR-2006-214298, NIA-NASA Langley, National Institute of Aerospace, Hampton, VA (2006)
3. Bostan, A.: Algorithmique efficace pour des opérations de base en Calcul formel. Ph.d. thesis (2003), <http://algo.inria.fr/bostan/these/These.pdf>
4. Chen, L., Miné, A., Cousot, P.: A Sound Floating-Point Polyhedra Abstract Domain. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 3–18. Springer, Heidelberg (2008)
5. Cohen, C.: Construction of Real Algebraic Numbers in CoQ. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 67–82. Springer, Heidelberg (2012)
6. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. SIGSAM Bull 10, 10–12 (1976)
7. Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In: IEEE Symposium on Computer Arithmetic, pp. 188–195 (2005)
8. Harrison, J.: Floating Point Verification in HOL. In: Schubert, E.T., Alves-Foss, J., Windley, P. (eds.) HUG 1995. LNCS, vol. 971, pp. 186–199. Springer, Heidelberg (1995)
9. IEEE. IEEE standard for binary floating-point arithmetic. Institute of Electrical and Electronics Engineers, New York, Note: Standard 754–1985 (1985)
10. Maddalon, J., Butler, R., Muñoz, C., Dowek, G.: Mathematical basis for the safety analysis of conflict prevention algorithms. Technical Memorandum NASA/TM-2009-215768, NASA, Langley Research Center, Hampton VA 23681-2199, USA (June 2009)



11. Martel, M.: Program transformation for numerical precision. In: PEPM, pp. 101–110 (2009)
12. Miner, P.S.: Defining the ieee-854 floating-point standard in pvs (1995)
13. Narkawicz, A., Muñoz, C., Dowek, G.: Formal verification of air traffic prevention bands algorithms. Technical Memorandum NASA/TM-2010-216706, NASA, Langley Research Center, Hampton VA 23681-2199, USA (June 2010)
14. Neron, P.: A formal proof of square root and division elimination in embedded programs - long version (2012), <http://www.lix.polytechnique.fr/~neron/>
15. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. Univ. of California Press (1951)
16. Weispfenning, V.: Quantifier elimination for real algebra - the quadratic case and beyond. Appl. Algebra Eng. Commun. Comput. 8(2), 85–101 (1997)

# Coherent and Strongly Discrete Rings in Type Theory\*

Thierry Coquand, Anders Mörtberg, and Vincent Siles

Department of Computer Science and Engineering  
University of Gothenburg, Sweden  
{coquand,mortberg,siles}@chalmers.se

**Abstract.** We present a formalization of coherent and strongly discrete rings in type theory. This is a fundamental structure in constructive algebra that represents rings in which it is possible to solve linear systems of equations. These structures have been instantiated with Bézout domains (for instance  $\mathbb{Z}$  and  $k[x]$ ) and Prüfer domains (generalization of Dedekind domains) so that we get certified algorithms solving systems of equations that are applicable on these general structures. This work can be seen as basis for developing a formalized library of linear algebra over rings.

**Keywords:** Formalization of mathematics, Constructive algebra, COQ, SSREFLECT.

## 1 Introduction

One of the fundamental operations in linear algebra is the ability to solve linear systems of equations. The concept of (strongly discrete) coherent rings abstracts over this ability which makes them an important notion in constructive algebra [14]. This makes these rings suitable as a basis for developing computational homological algebra, that is, linear algebra over rings instead of fields [3].

Another reason that these rings are important in constructive algebra is that they generalize the notion of Noetherian rings [1]. Classically any Noetherian ring is coherent but the situation in constructive mathematics is more complex and there is in fact no standard constructive definition of Noetherianity [16]. Logically, Noetherianity is expressed by a higher-order condition (it involves quantification over every ideal of the ring) while "coherent" is a simpler notion, which involves only quantification on matrices over the ring, and "strongly discrete" is a first-order notion.

One important example (aside from fields) of coherent strongly discrete rings are Bézout domains, which are a non-Noetherian generalization of principal ideal domains (rings where all ideals are generated by one element). The two standard

---

\* The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

<sup>1</sup> Rings where all ideals are finitely generated.

examples of Bézout domains are  $\mathbb{Z}$  and  $k[x]$  where  $k$  is a field. Another example of coherent strongly discrete rings are Prüfer domains with decidable divisibility which are a non-Noetherian generalization of Dedekind domains. The condition of being a Prüfer domain captures what Dedekind thought was the most important property of Dedekind domains [1], namely the ability to invert ideals (which is usually hidden in standard classical treatments of Dedekind domains). This property also has applications in control theory [18].

All our proofs and definitions are expressed in a constructive framework. While it would be possible to use classical logic in the proof of correctness of our algorithms, we feel that they are clearer and shorter in this way. It can also be argued that our definitions are better expressed in this way. For instance, we can define a coherent ring as one for which a linear system has a finite number of generators. In a classical framework, to express this in a computationally meaningful way would involve the notion of recursive functions.

All of these notions have been formalized<sup>2</sup> using the SSREFLECT extension [11] to the COQ proof assistant [5]. This work can be seen as a generalization of the previous formalization of linear algebra in the SSREFLECT library [10].

The main motivation behind this work is that it can be seen as a basis for a formalization of computational homological algebra. This approach is inspired by the one of HOMALG [3] where homological algorithms (without formalized correctness proofs) are implemented based on a notion that they call *computable rings* [2] which in fact are the same as coherent strongly discrete rings. Another source of inspiration is the work of Lombardi and Quitté [13] on constructive commutative algebra.

This paper is organized as follows: first the formalization of coherent rings is presented followed by strongly discrete rings. Next Prüfer domains are explained together with the proofs that they are both coherent and strongly discrete. This is followed by a section on how to implement a computational version of the SSREFLECT development. We end by a section on conclusions and further work.

## 2 Coherent Rings

Given a ring  $R$  (in our setting commutative but it is possible to consider non-commutative rings as well [2]) one important problem to study is how to solve linear systems over  $R$ . If  $R$  is a field, then we have a nice description of the space of solution by a basis of solutions. Over an arbitrary ring  $R$  there is in general no basis<sup>3</sup>. But an important weaker property is that there is a finite number of solutions which generate all solutions. We say that the ring is *coherent* if this is the case.

<sup>2</sup> Documentation and formalization can be found at:

<http://www.cse.chalmers.se/~mortberg/coherent/>

<sup>3</sup> For instance over the ring  $R = k[X, Y, Z]$  where  $k$  is a field, the equation  $pX + qY + rZ = 0$  has no basis of solutions. It can be shown that a generating system of solutions is given by  $(-Y, X, 0)$ ,  $(Z, 0, -X)$ ,  $(0, -Z, Y)$ .

More concretely, given a rectangular matrix  $M$  over  $R$  we want to find a finite number of solutions  $X_1, \dots, X_n$  of the system  $MX = 0$  such that any solution is of the form  $a_1X_1 + \dots + a_nX_n$  where  $a_1, \dots, a_n \in R$ . If this is possible, we say that the module of solutions of the system  $MX = 0$  is finitely generated. This can be reformulated with matrices: we want to find a matrix  $L$  such that

$$MX = 0 \leftrightarrow \exists Y.X = LY$$

A ring is *coherent* if for any matrix  $M$  it is possible to compute a matrix  $L$  such that this holds. If this is the case it follows that  $ML = 0$ .

For this it is enough to consider the case where  $M$  has only one line. Indeed, assume that for any  $1 \times n$  matrix  $M$  we can find a  $n \times m$  matrix  $L$  such that  $MX = 0$  iff  $X = LY$  for some  $Y$ . To solve the system

$$M_1X = \dots = M_kX = 0$$

where each  $M_i$  is a  $1 \times n$  matrix first compute  $L_1$  such that  $M_1X = 0$  iff  $X = LY_1$  for some  $Y_1$ . Next compute  $L_2$  such that  $M_2L_1Y_1 = 0$  iff  $Y_1 = L_2Y_2$ . At the end we obtain  $L_1, \dots, L_k$  such that  $M_1X = \dots = M_kX = 0$  iff  $X$  is of the form  $L_1 \dots L_k Y$  and so  $L_1 \dots L_k$  provide a system of generators for the solution of the system.

Hence it is sufficient to formulate the condition for coherent rings as: For any row matrix  $M$  it is possible to find a matrix  $L$  such that

$$MX = 0 \leftrightarrow \exists Y.X = LY$$

Note that the notion of coherent is not stressed in classical presentations of algebra since Noetherian rings are automatically coherent, but in a computationally meaningless way. It is however fundamental, both conceptually [13,14] and computationally. The system HOMALG [3] for instance takes this notion as the central one.

In the development, coherent rings have been implemented as in [9] using the **Canonical Structure** mechanism of COQ. In the SSREFLECT libraries matrices are represented by finite functions over pairs of ordinals (the indices):

```
(* 'I_n *)
Inductive ordinal (n : nat) := Ordinal m of m < n.

(* 'M[R]_(m,n) = 'M_(m,n) *)
(* 'rV[R]_m = 'M[R]_(1,m) *)
(* 'cV[R]_m = 'M[R]_(m,1) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Hence the sizes of the matrices need to be known when implementing coherent rings. But in general the size of  $L$  cannot be predicted so we need an extra function that computes this:

```

Record mixin_of (R : ringType) : Type := Mixin {
  size_solve : forall m, 'rV[R]_m -> nat;
  solve_row : forall m (V : 'rV[R]_m), 'M[R]_(m,size_solve V);
  _ : forall m (V : 'rV[R]_m) (X : 'cV[R]_m),
    reflect (exists Y : 'cV[R]_(size_solve V), X = solve_row V *m Y)
      (V *m X == 0)
}.

```

Here  $*m$  denotes matrix multiplication and  $V *m X == 0$  is the boolean equality of matrices, so the specification says that this equality is reflected by the existence statement. An alternative to having a function computing the size would be to output a dependent pair but this has the undesired behavior that the pair has to be destructed when stating lemmas about it which in turn would mean that these lemmas would be cumbersome to use as it would not be possible to rewrite with them directly.

Using this we have implemented the algorithm for computing the generators of a system of equations:

```

Fixpoint solveMxN (m n : nat) :
  forall (M : 'M_(m,n)), 'M_(n,size_solveMxN M) :=
  match m return forall M : 'M_(m,n), 'M_(n,size_solveMxN M) with
  | S p => fun (M : 'M_(1 + _,n)) =>
    let L1 := solve_row (usubmx M)
    in L1 *m solveMxN (dsubmx M *m L1)
  | _ => fun _ => 1%:M
end.

```

```

Lemma solveMxNP : forall m n (M : 'M[R]_(m,n)) (X : 'cV[R]_n),
  reflect (exists Y : 'cV_(size_solveMxN M), X = solveMxN M *m Y)
    (M *m X == 0).

```

In order to instantiate this structure one can of course directly give an algorithm that computes the solution of a single row system. However there is another approach that will be used in the rest of the paper that is based on the intersection of finitely generated ideals.

## 2.1 Ideal Intersection and Coherence

In the case when  $R$  is an integral domain one way to prove that  $R$  is coherent is to show that the intersection of two finitely generated ideals is again finitely generated. This amounts to given two ideals  $I = (a_1, \dots, a_n)$  and  $J = (b_1, \dots, b_m)$  compute generators  $(c_1, \dots, c_k)$  of  $I \cap J$ . For  $I \cap J$  to be the intersection of  $I$  and  $J$  it should satisfy

$$\begin{aligned}
 I \cap J &\subseteq I \\
 I \cap J &\subseteq J \\
 \forall x. x \in I \wedge x \in J &\rightarrow x \in I \cap J
 \end{aligned}$$

The first two of these mean that the generators of  $I \cap J$  should be possible to write as a linear combination of the generators of both  $I$  and  $J$ . The third property states that if  $x$  can be written as a linear combination of the generators of  $I$  and  $J$  then it can be written as a linear combination of the generators of  $I \cap J$ .

A convenient way to express this in COQ is to use strongly discrete rings, which are discussed in section 3. For now we just assume that we can find matrices  $V$  and  $W$  such that  $IV = I \cap J$  and  $JW = I \cap J$  (with matrix multiplication and ideals represented by row-vectors containing the generators). Using this there is an algorithm to compute generators of the solutions of a system:

$$m_1x_1 + \dots + m_nx_n = 0$$

The main idea is to compute generators,  $M_0$ , of the solution for  $m_2x_2 + \dots + m_nx_n = 0$  by recursion and also compute generators  $t_1, \dots, t_p$  of  $(m_1) \cap (-m_2, \dots, -m_n)$  together with  $V$  and  $W$  such that

$$\begin{aligned} (m_1)V &= (t_1, \dots, t_p) \\ (-m_2, \dots, -m_n)W &= (t_1, \dots, t_p) \end{aligned}$$

The generators of the module of solutions are then given by:

$$\begin{bmatrix} V & 0 \\ W & M_0 \end{bmatrix}$$

This has been implemented by:

```
Fixpoint solve_int m : forall (M : 'rV_m), 'M_(m,size_int M) :=
  match m return forall (M : 'rV_m), 'M_(m,size_int M) with
  | S p => fun (M' : 'rV_(1 + p)) =>
    let m1 := lsubmx M' in
    let ms := rsubmx M' in
    let MO := solve_int ms in
    let V := cap_wl m1 (-ms) in
    let W := cap_wr m1 (-ms) in
    block_mx (if m1 == 0 then delta_mx 0 0 else V) 0
              (if m1 == 0 then 0 else W) MO
  | 0 => fun _ => 0
  end.
```

```
Lemma solve_intP : forall m (M : 'rV_m) (X : 'cV_m),
  reflect (exists Y : 'cV[R]_(size_int M), X = solve_int M *m Y)
  (M *m X == 0).
```

Here `cap_wl` computes  $V$  and `cap_wr` computes  $W$ . Note that some special care has to be taken if  $m_1$  is zero, if this is the case we output a matrix:

$$\begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & \dots & 0 & M_0 \end{bmatrix}$$

However it would be desirable to output just

$$\begin{bmatrix} 1 & 0 \\ 0 & M_0 \end{bmatrix}$$

But this would not have the correct size. This could be solved by having a more complicated function that outputs a sum type with matrices of two different sizes. As this would give slightly more complicated proofs we decided to pad with zeroes instead. In section 5 we will discuss how to implement a more efficient algorithm, without any padding, that is more suitable for computation.

### 3 Strongly Discrete Rings

An important notion in constructive mathematics is the notion of *discrete* ring, that is, rings with decidable equality. Another important notion is *strongly discrete* rings, these are rings where membership in finitely generated ideals is decidable and if  $x \in (a_1, \dots, a_n)$  there is an algorithm computing  $w_1, \dots, w_n$  such that  $x = \sum_i a_i w_i$ .

Examples of such rings are multivariate polynomial rings over discrete fields (via Gröbner bases [6][2]) and Bézout domains with explicit divisibility, that is, whenever  $a \mid b$  one can compute  $x$  such that  $b = xa$ . We have represented strongly discrete rings in COQ as:

```
CoInductive member_spec (R : ringType) n (x : R) (I : 'rV[R]_n)
  : option 'cV[R]_n -> Type :=
| Member J of x%M = I *m J : member_spec x I (Some J)
| NMember of (forall J, x%M != I *m J) : member_spec x I None.
```

```
Record mixin_of (R : ringType) : Type := Mixin {
  member : forall n, R -> 'rV_n -> option 'cV_n;
  _ : forall n (x : R) (I : 'rV_n), member_spec x I (member x I)
}.
```

The structure of strongly discrete rings contains a function taking an element and a row vector (with the generators of the ideal) and return an option type with a column vector. This is `Some J` if  $x$  can be written as  $I *m J$  and if it is `None` then there should also be a proof that there cannot be any  $J$  satisfying  $x = I *m J$ . Note that the use of `CoInductive` has nothing to do with coinduction but it should be seen as a datatype without any recursion schemes (as opposed to datatypes defined using `Inductive`) on which one can do case-analysis, for more information see [11].

#### 3.1 Ideal Theory

In the development we have chosen to represent finitely generated ideals as row vectors, so an ideal in  $R$  with  $n$  generators is represented as a row matrix of type

'rV[R]\_n. This way operations on ideals can be implemented using functions on matrices and properties can be proved using the matrix library of SSREFLECT.

A nice property of strongly discrete rings is that the inclusion relation of finitely generated ideals is decidable. This means that we can decide if  $I \subseteq J$  and if this is the case express every generator of  $I$  as a linear combination of the generators of  $J$ . We have implemented this as the function `subid` with correctness expressed as:

**Notation** "A <= B" := (subid A B).

**Notation** "A == B" := ((A <= B) && (B <= A)).

**Lemma** `subidP` : forall m n (I : 'rV[R]\_m) (J : 'rV[R]\_n),  
 reflect (exists D, I = J \*m D) (I <= J)%IS.

Note that this is expressed using matrix multiplication, so `subidP` says that if  $I \leq J$  then every generator of  $I$  can be written as a linear combination of generators of  $J$ .

Ideal multiplication is an example where it is convenient to represent ideals as row vectors. As the product of two finitely generated ideals is generated by all products of generators of the ideals this can be expressed compactly using matrix operations:

**Definition** `mulid` m n (I : 'rV\_m) (J : 'rV\_n) : 'rV\_(m \* n) :=  
 mxvec (I<sup>T</sup> \*m J).

**Notation** "I \*i J" := (mulid I J).

Here `mxvec` flattens a matrix of type 'M[R]\_(m,n) into a row vector of type 'rV[R]\_(m \* n) and  $I^T$  is the transpose of  $I$ . By representing ideals as row vectors we get compact definitions and quite simple proofs as the theory already developed about matrices can be used when proving properties of ideal operations.

It is also convenient to specify what the intersection of  $I$  and  $J$  is: it is an ideal  $K$  such that  $K \leq I$ ,  $K \leq J$  and forall (x : R), member x I -> member x J -> member x K. So in order to prove that an integral domain is coherent it suffices to give an algorithm that computes  $K$  and prove that it satisfies these three properties. The `cap_wr` and `cap_wl` functions used in `solve_with_int` can then be implemented easily by explicitly computing  $D$  in `subidP`.

### 3.2 Coherent Strongly Discrete Rings

If a ring  $R$  is both coherent and strongly discrete it is not only possible to solve homogeneous systems  $MX = 0$  but also any system  $MX = A$ . The algorithm for computing this is expressed by induction on the number of equations where the case of one equation follow directly from the fact that the ring is strongly discrete. In the other case the matrix looks like:

$$\begin{bmatrix} R_1 \\ M \end{bmatrix} X = \begin{bmatrix} a_1 \\ A \end{bmatrix}$$



Now compute generators  $G_1$  for the module of system of solutions of the homogeneous system  $R_1X = 0$  and also test if  $a_1 \in R_1$ , if this is not the case the system is not solvable otherwise get  $W_1$  such that  $R_1W_1 = a_1$ . Now compute by recursion the solution  $S$  of  $MG_1X = A - MW_1$  such that  $MG_1S = A - MW_1$ . The solution to the system is then  $W_1 + G_1S$  as

$$\begin{bmatrix} R_1 \\ M \end{bmatrix} (W_1 + G_1S) = \begin{bmatrix} R_1W_1 + R_1G_1S \\ MW_1 + MG_1S \end{bmatrix} = \begin{bmatrix} a_1 \\ A \end{bmatrix}$$

This algorithm has been implemented and proved correct as the function `solve_general`. Together with `solveMxN` this constitutes the only operations used as basis in the libraries of the HOMALG project [3].

### 3.3 Bézout Domains Are Strongly Discrete and Coherent

The first example of coherent strongly discrete rings that we studied were Bézout domains with explicit divisibility. These are integral domains where every finitely generated ideal is principal (generated by a single element). The two main examples of Bézout domains are  $\mathbb{Z}$  and  $k[x]$  where  $k$  is a discrete field.

Bézout domains can also be characterized as rings with a GCD operation in which there is a function computing the elements of the Bézout identity. This means that given  $a$  and  $b$  one can compute  $x$  and  $y$  such that  $xa + by$  is associate<sup>4</sup> to  $\gcd(a, b)$ . Based on this it is straightforward to implement a function that given a finitely generated ideal  $(a_1, \dots, a_n)$  computes  $g$  such that  $(a_1, \dots, a_n) \subseteq (g)$  and  $(g) \subseteq (a_1, \dots, a_n)$  where this  $g$  is the greatest common divisor of all the  $a_i$ . To test if  $x \in (a_1, \dots, a_n)$  in a Bézout domain first compute a principal ideal  $(g)$  and then test if  $g \mid x$  and if this is the case we we can construct the witness and otherwise we know that  $x \notin (a_1, \dots, a_n)$ .

For showing that Bézout domains are coherent let  $I$  and  $J$  be two finitely generated ideals and compute principal ideals such that  $I = (a)$  and  $J = (b)$ . Now it easy to prove that  $I \cap J = (\text{lcm}(a, b))$ , where  $\text{lcm}(a, b)$  is the lowest common multiple of  $a$  and  $b$  which is computable in our setting as any Bézout ring is a GCD domain with explicit divisibility. Hence we have now proved that  $\mathbb{Z}$  and  $k[x]$  are both coherent and strongly discrete which means that we can solve arbitrary systems of equations over them.

## 4 Prüfer Domains

Another class of rings that are coherent are *Prüfer domains*. These can be seen as non-Noetherian analogues of Dedekind domains and have many different characterizations, for instance does Bourbaki list fourteen of them [4]. The one we choose here is the one in [13] that says that a Prüfer domain is an integral domain where given any  $x$  and  $y$  there exists  $u, v$  and  $w$  such that

<sup>4</sup>  $a$  and  $b$  are associates if  $a \mid b$  and  $b \mid a$  or equivalently that there exists a unit  $u \in R$  such that  $a = bu$ .

$$ux = vy$$

and

$$(1 - u)y = wx$$

This is implemented in COQ by:

```
Record mixin_of (R : ringType) : Type := Mixin {
  prufer: R -> R -> (R * R * R)%type;
  _ : forall x y, let: (u,v,w) := prufer x y in
    u * x = v * y /\ (1 - u) * y = w * x
}.
```

As we require that Prüfer domains have explicit divisibility it is possible to prove that they are strongly discrete which in turn means that we can use the library of ideal theory developed for strongly discrete rings when proving that they are coherent. However it would be possible to prove that Prüfer domains are coherent without assuming explicit divisibility.

The most basic examples of Prüfer domains are Bézout domains (in particular  $\mathbb{Z}$  and  $k[x]$ ). However there are many other examples, for instance if  $R$  is a Bézout domain then the ring of elements integral over  $R$  is a Prüfer domain, this gives examples from algebraic geometry like  $k[x, y]/(y^2 + x^4 - 1)$  and algebraic number theory like  $\mathbb{Z}[\sqrt{-5}]$ .

### 4.1 Principal Localization Matrices and Strong Discreteness

The key algorithm in the proof that Prüfer domains with explicit divisibility are both strongly discrete and coherent is an algorithm computing a *principal localization matrix* of an ideal  $\mathfrak{S}$ . This means that given a finitely generated ideal  $(x_1, \dots, x_n)$  one can compute a  $n \times n$  matrix  $M = (a_{ij})$  such that:

$$\sum_{i=1}^n a_{ii} = 1$$

and

$$\forall ij. a_{ij}x_i = a_{li}x_j$$

Note that there is no constraint  $n \neq 0$  which means that the first of these is a bit problematic as if  $n = 0$  the sum will be empty and hence 0. To remedy this we express the property formally as:

```
Definition P1 n (M : 'M[R]_n) :=
  \big[+%R/0]_(i: 'I_n) (M i i) = (0 < n)%:R.
```

```
Definition P2 n (I : 'rV[R]_n) (M : 'M[R]_n) :=
  forall (i j l : 'I_n), (M l j) * (I 0 i) = (M l i) * (I 0 j).
```

```
Definition isPLM n (I : 'rV[R]_n) (M : 'M[R]_n) := P1 M /\ P2 I M.
```

The first statement uses an implicit coercion from booleans to rings where `false` is coerced to 0 and `true` to 1. The algorithm computing a principal localization matrix, `plm`, is quite involved so we have omitted it from this presentation, the interested reader should have a look in the formal development and at the proofs in [8] and [13]. We have proved that this algorithm satisfies the above specification:

**Lemma** `plmP` : `forall n (I : 'rV[R]_n), isPLM I (plm I)`.

The reason that principal localization matrices are interesting is that they give a way to compute the *inverse of a finitely generated ideal*  $I$ , this is a finitely generated ideal  $J$  such that  $IJ$  (with ideal multiplication) is principal. In fact if  $I = (x_1, \dots, x_n)$  and  $M = (a_{ij})$  is its principal localization matrix then every column of  $M$  is an inverse to  $I$ . This means that we can define an algorithm for computing the inverse of ideals in Prüfer domains:

**Definition** `inv_id n` : `'I_n -> 'rV[R]_n -> 'rV[R]_n := match n with`  
`| S p => fun (i : 'I_(1 + p)%N) (I : 'rV[R]_(1 + p)%N) =>`  
`(col i (plm I))^T`  
`| _ => fun _ _ => 0`  
`end.`

**Lemma** `inv_idP n (I : 'rV[R]_n) i` :  
`(inv_id i I * i I == (I 0 i)%:M)%IS.`

Using this it is possible to prove that Prüfer domains with explicit divisibility are strongly discrete. To compute if  $x \in I$  first compute  $J$  such that  $IJ = (a)$ . Now  $x \in I$  iff  $(x) \subseteq I$  iff  $xJ \subseteq (a)$ . This can be decided if we can decide when an element is divisible by  $a$ .

We have used this to prove that our implementation of Prüfer domains is strongly discrete which means that the theory about ideals developed for strongly discrete rings can be used when proving that they are coherent.

## 4.2 Coherence

The key property of ideals in Prüfer domains for computing the intersection is that finitely generated ideals  $I$  and  $J$  satisfy:

$$(I + J)(I \cap J) = IJ$$

This means that we can devise an algorithm for computing generators for the intersection by first computing  $(I + J)^{-1}$  such that  $(I + J)^{-1}(I + J) = (a)$  and then get that

$$I \cap J = \frac{(I + J)^{-1}IJ}{a}$$

Note the use of division here. In fact it is possible to compute the intersection without assuming division but then the algorithm is more complicated. Using this the function for computing generators of the intersection is:

```

Definition pcap (n m : nat) (I : 'rV[R]_n) (J : 'rV[R]_m) :
  'rV[R]_(pcap_size I J).+1 := match find_nonzero (I +i J) with
  | Some i => let sIJ := I +i J in
              let a := sIJ 0 i in
              let acap := inv_id i sIJ *i I *i J in
              (0 : 'M_1) +i (\row_i (odflt 0 (acap 0 i %/? a)))
  | None => 0
end.

```

Here `%/?` is the explicit divisibility function of  $R$ . The reason to add 0 as a generator of the ideal is simply to have the correct size as the formalized proof that  $R$  is coherent if  $I \cap J$  is computable requires that  $I \cap J$  is nonempty. Also note the function `find_nonzero` which finds the first nonzero element in a row-vector. This could have been implemented using the `pick` function for picking an element satisfying a decidable predicate which is provided for all `SSREFLECT` rings. But in order to simplify the translation to an efficient version of the algorithm we avoid using it here.

To prove that `pcap` really computes the intersection we need to first prove the main property used above for finding the algorithm computing  $I \cap J$ :

```

Lemma pcap_id (n m : nat) (I : 'rV[R]_n) (J : 'rV[R]_m) :
  ((I +i J) *i pcap I J == I *i J)%IS.

```

Using this it is possible to prove that `pcap` compute the intersection:

```

Lemma pcap_subidl m n (I : 'rV_m) (J : 'rV_n): (pcap I J <= I)%IS.

```

```

Lemma pcap_subidr m n (I : 'rV_m) (J : 'rV_n): (pcap I J <= J)%IS.

```

```

Lemma pcap_member m n x (I : 'rV[R]_m) (J : 'rV[R]_n) :
  member x I -> member x J -> member x (pcap I J).

```

Hence we have now proved that Prüfer domains with explicit divisibility are coherent strongly discrete rings so not only can we solve homogeneous systems over them but also any linear system of equations.

### 4.3 Examples of Prüfer Domains

As mentioned before, any Bézout domain is a Prüfer domain. The proof of this is easy:

```

Definition bezout_calc (x y: R) : (R * R * R)%type :=
  let: (g,c,d,a,b) := egcdr x y in (d * b, a * d, b * c).

```

```

Lemma bezout_calcP (x y : R) : let: (u,v,w) := bezout_calc x y in
  u * x = v * y /\ (1 - u) * y = w * x.

```

Here `egcdr` is the extended Bézout algorithm where  $g$  is the *gcd* of  $x$  and  $y$ ,  $x = ag$ ,  $y = bg$  and  $ca + db = 1$ .

We have not yet formalized the proof that  $\mathbb{Z}[\sqrt{-5}]$  and  $k[x,y]/(y^2 - 1 + x^4)$  are Prüfer domains but we have previously implemented this in HASKELL [15].

## 5 Computations

In the paper algorithms are presented on structures using rich dependently typed datatypes which is convenient when proving properties but for computation this is not necessary. In fact it can be more efficient to implement the algorithms on simply typed datatypes instead, a good example is matrices: As explained in section 2 they are represented using finite functions from the indices (represented using ordinals) but this representation is not suitable for computation as finite functions are represented by their graph which has to be traversed linearly each time the function is evaluated.

In order to develop more efficient versions of the algorithms we use a previously developed library where matrices are represented using lists of lists and implement the algorithms on this representation. These algorithms are then linked to the inefficient versions using translation lemmas. The methodology that we follow is summarized in 7 as:

1. Implement an abstract version of the algorithm using SSREFLECT structures and use the libraries to prove properties about them. Here we can use the full power of dependent types when proving correctness.
2. Refine this algorithm into an efficient one using SSREFLECT structures and prove that it behaves like the abstract version.
3. Translate the SSREFLECT structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

So far we have only presented step 1. The second step involves giving more efficient algorithms, a good example of this is the algorithms on ideals. A simple optimization that can be made is to ensure that there are no zeroes as generators in the output of the ideal operations. The goal would then be to prove that the more efficient operations generate the same ideal as the original operation. Another example is `solve_int` that can be implemented without padding with zeroes, this would then be proved to produce a set of solution of the system and then be translated to a more efficient algorithm on list based matrices.

The final step corresponds to implementing “computable” counterparts of the structures that we presented so far based on simple types. For example, computable coherent rings are implemented as:

```
Record mixin_of (R : coherentRingType)
  (CR : cstronglyDiscreteType R) : Type := Mixin {
  csize_solve : nat -> seqmatrix CR -> nat;
  csolve_row : nat -> seqmatrix CR -> seqmatrix CR;
  _ : forall n (V : 'rV[R]_n),
    seqmx_of_mx CR (solve_row V) = csolve_row n (seqmx_of_mx _ V);
  _ : forall n (V : 'rV[R]_n),
    size_solve V = csize_solve n (seqmx_of_mx _ V)
}.
```

Here `seqmatrix` is the list based representation of matrices and `seqmx_of_mx` is the translation function from SSREFLECT matrices to `seqmatrix`. Using this more efficient versions of the algorithms presented above can be implemented simply by changing the functions on SSREFLECT matrices to functions on `seqmatrix`:

```
Fixpoint csolveMxN m n (M : seqmatrix CR) : seqmatrix CR :=
  match m with
  | S p => let u := usubseqmx 1 M in
           let d := dsubseqmx 1 M in
           let G := csolve_row n u in
           let k := csize_solve n u in
           let R := mulseqmx n k d G in
           mulseqmx k (csize_solveMxN p k R) G (csolveMxN p k R)
  | _ => seqmx1 CR n
end.
```

```
Lemma csolveMxNE : forall m n (M : 'M[R]_(m,n)),
  seqmx_of_mx _ (solveMxN M) = csolveMxN m n (seqmx_of_mx _ M).
```

The lemma states that solving the system on SSREFLECT matrices and then translating is the same as first translating and then compute the solution using the list based algorithm. The proof of this is straightforward as all of the functions in the algorithm have translation lemmas, so it is done by expanding definitions and translating using already implemented translation lemmas.

This way we have implemented all of the above algorithms and instances to make some computations with  $\mathbb{Z}$  using the algorithms for Bézout domains. First we can compute the generators of  $(2) \cap (3, 6)$ :

```
Eval vm_compute in (cbcap 1 2 [::[::2]] [::[::3; 6]]).
= [:: [:: 6]]
```

Next we can test if  $6 \in (2)$ :

```
Eval vm_compute in (cmember 1%N 6 [::[:: 2]]).
= Some [:: [:: 3]]
```

It is also possible to solve the homogeneous system:

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
Eval vm_compute in (csolveMxN 2 2 [::[:: 1;2];[::2;4]]).
= [:: [:: 2; 0];
   [:: -1; 0]]
```

and the inhomogeneous system:

$$\begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

```

Eval vm_compute in (csolveGeneral 2 2 [::[:: 2; 3]; [:: 4; 6]]
                    [::[:: 4]; [:: 8]]).
= Some [:: [:: -4];
        [:: 4]]

```

The system  $2x = 1$  does not have a solution in  $\mathbb{Z}$ :

```

Eval vm_compute in (csolveGeneral 1 1 [::[:: 2]] [::[::1]]).
= None

```

We can also do some computations on the algorithms for Prüfer domains using  $\mathbb{Z}$ :

```

Eval vm_compute in (cplm 3 [::[:: 2; 3; 5]]).
= [:: [:: 8; 12; 20];
   [:: 12; 18; 30];
   [:: -10; -15; -25]]

```

```

Eval vm_compute in (cinv_id 2 0 [:: [:: 2; 3]]).
= [:: [:: -2; 2]]

```

The first computation computes the principal localization matrix of  $(2, 3, 5)$  and the second compute the inverse of the ideal  $(2, 3)$ .

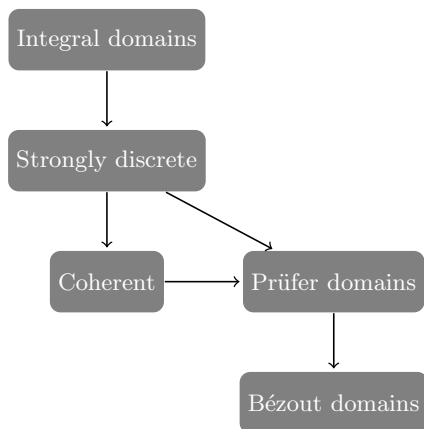
## 6 Conclusions and Further Work

In this paper we have represented in type theory interesting and mathematically nontrivial results in constructive algebra. The algorithms based on coherent and strongly discrete rings have been refined to more efficient algorithms on simple datatypes, this way we get certified mathematical algorithms that are suitable for computation. Hence can this work be seen as an example that the methodology presented in [7] is applicable on more complicated structures as well.

In the future it would be interesting to prove that multivariate polynomial rings over discrete fields are coherent and strongly discrete. This would require a formalization of Gröbner bases and the Buchberger algorithm which has already been done in COQ [17,19]. It would be interesting to reimplement this using SSREFLECT and compare the complexity of the formalizations.

It would also be interesting to use this work as a basis for a library of formalized computational homological algebra inspired by the HOMALG project. In fact `solveMxN` and `solveGeneral` are the only operations used as a basis in HOMALG [2]. This formalization would involve first proving that the category of finitely presented modules over coherent strongly discrete rings form an abelian category and then use this to implement further algorithms for doing homological computations.

In SSREFLECT all rings are equipped with a choice operator which can be used to pick an element satisfying a decidable predicate. This could have been used more in our development, for instance in the implementation of `pcap` to find a nonzero generator of an ideal. We believe that using this feature more



**Fig. 1.** The extension to the SSREFLECT hierarchy

would lead to simpler formal proofs, but our experience is that the use of choice complicates the implementation of efficient algorithms. As we want to be able to compute with our algorithms inside COQ we decided to have slightly more complicated proofs but easier translation to efficient algorithms.

In Fig. 1 the extension to the SSREFLECT hierarchy is presented. Integral domains are already present in the hierarchy and the extension consists of the other structures. The arrows represent that the target is an instance of the source. This presentation differs from standard presentations in constructive algebra [13,14] as there is no need to assume that coherent rings and Prüfer domains are strongly discrete. The motivation behind this design choice is that it simplified the formalization and the examples that we are primarily interested in are all strongly discrete anyway. We actually started to formalize the notions without assuming that the rings are strongly discrete but this led to too complicated formal proofs as we could not use the library of ideal theory developed for strongly discrete rings. However, in constructive algebra ideal theory is usually developed without assuming decidable ideal membership, but in our experience, both the SSREFLECT library and tactics are best suited for theories with decidable functions.

A consequence of this is that it is difficult to formalize things in full generality, for instance are all rings assumed to be not only strongly discrete but also discrete. It would be more natural from the point of view of constructive mathematics to represent more general structures. However, while the use of SSREFLECT imposes some decidability conditions, we found that in this framework of decidable structures the notations and tactics provided are particularly elegant and well-suited.



## References

1. Avigad, J.: Methodology and metaphysics in the development of Dedekind's theory of ideals. In: *The Architecture of Modern Mathematics. Essays in History and Philosophy*. Oxford University Press, Oxford (2006)
2. Barakat, M., Lange-Hegermann, M.: An axiomatic setup for algorithmic homological algebra and an alternative approach to localization. *J. Algebra Appl.* 10(2), 269–293 (2011)
3. Barakat, M., Robertz, D.: HOMALG – A Meta-Package for Homological Algebra. *J. Algebra Appl.* 7(3), 299–317 (2008)
4. Bourbaki, N.: *Commutative algebra. Elements of Mathematics*, ch. 1-7. Springer (1998)
5. COQ development team. *The COQ Proof Assistant Reference Manual*, version 8.3. Technical report (2010)
6. Cox, D., Little, J., O'Shea, D.: *Ideals, Varieties and Algorithms: An introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer (2006)
7. Dénès, M., Mörtberg, A., Siles, V.: A Refinement-Based Approach to Computational Algebra in CoQ. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 83–98. Springer, Heidelberg (2012)
8. Ducos, L., Lombardi, H., Quitté, C., Salou, M.: Théorie algorithmique des anneaux arithmétiques, des anneaux de Prüfer et des anneaux de Dedekind. *Journal of Algebra* 281(2), 604–650 (2004)
9. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009)
10. Gonthier, G.: Point-Free, Set-Free Concrete Linear Algebra. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 103–118. Springer, Heidelberg (2011)
11. Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA (2009)
12. Lombardi, H., Perdry, H.: The Buchberger Algorithm as a Tool for Ideal Theory of Polynomial Rings in Constructive Mathematics (1998)
13. Lombardi, H., Quitté, C.: *Algèbre commutative, Méthodes constructives: Modules projectifs de type fini*. Calvage et Mounet (2011)
14. Mines, R., Richman, F., Ruitenburg, W.: *A Course in Constructive Algebra*. Springer (1988)
15. Mörtberg, A.: *Constructive Algebra in Functional Programming and Type Theory*. Master's thesis, Chalmers University of Technology (2010)
16. Perdry, H., Schuster, P.: Noetherian orders. *Mathematical. Structures in Comp. Sci.* 21(1), 111–124
17. Persson, H.: *An Integrated Development of Buchberger's Algorithm in Coq* (2001)
18. Quadrat, A.: The fractional representation approach to synthesis problems: An algebraic analysis viewpoint part ii: Internal stabilization. *SIAM J. Control Optim.* 42(1), 300–320 (2003)
19. Théry, L.: A Certified Version of Buchberger's Algorithm. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 349–364. Springer, Heidelberg (1998)

# Improving Real Analysis in Coq: A User-Friendly Approach to Integrals and Derivatives\*

Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond

Inria Saclay Île-de-France, ProVal, Palaiseau, F-91120  
LRI, Université Paris-Sud, CNRS, Orsay, F-91405  
{Sylvie.Boldo,Catherine.Lelay,Guillaume.Melquiond}@inria.fr

**Abstract.** Verification of numerical analysis programs requires dealing with derivatives and integrals. High confidence in this process can be achieved using a formal proof checker, such as Coq. Its standard library provides an axiomatization of real numbers and various lemmas about real analysis, which may be used for this purpose. Unfortunately, its definitions of derivative and integral are unpractical as they are partial functions that demand a proof term. This proof term makes the handling of mathematical formulas cumbersome and does not conform to traditional analysis. Other proof assistants usually do not suffer from this issue; for instance, they may rely on Hilbert's epsilon to get total operators. In this paper, we propose a way to define total operators for derivative and integral without having to extend Coq's standard axiomatization of real numbers. We proved the compatibility of our definitions with the standard library's in order to leverage existing results. We also greatly improved automation for real analysis proofs that use Coq standard definitions. We exercised our approach on lemmas involving iterated partial derivatives and differentiation under the integral sign, that were missing from the formal proof of a numerical program solving the wave equation.

## 1 Introduction

From Newton and Leibniz during the 17th century, many mathematicians have used integrals and derivatives. Their use is both for pure analysis theorems, but also more recently for applied mathematics. For example, numerical analysis aims at solving ordinary differential equations and partial differential equations. When the solutions are not analytic, it provides algorithms to approximate these solutions and bounds to assert their correctness. Typically, it consists in a numerical scheme over a discrete grid and its convergence proof, meaning that the approximation improves when the grid size decreases.

Recent advances in formal proof assistants have shown that they can be applied to various kinds of problems, but analysis and especially numerical analysis was

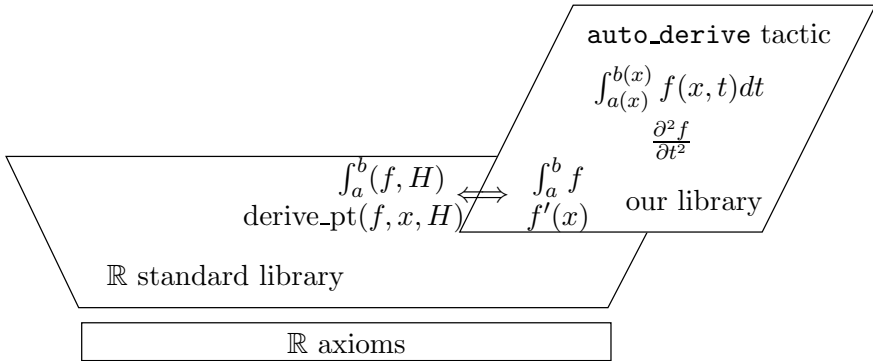
---

\* This research was supported by the F $\hat{e}$ st project (ANR-08-BLAN-0246-01) of the French national research organization (ANR) and by the Coquelicot project of the Digiteo cluster and the Île-de-France regional council.

not as much studied as algebra. One reason may be that formalizations of analysis were done years ago and seldom used. This is precisely the case in the standard library of Coq: derivatives and integrals were defined with real numbers a dozen years ago, but the libraries did not evolve with Coq. A more extensive use could have proved the ponderousness of the library. The most blatant example is that derivatives require a proof term to be written. This means that, instead of  $f'(x)$  we have to handle  $(f, x, H)$  where  $H$  is a proof that  $f$  is differentiable in  $x$ . This makes the rewritings clumsy and unpractical. More, this is not the way mathematicians prove their theorems: proofs that functions are regular enough, when present, are side-proofs that do not arise in the main development. Another example is the missing definitions and lemmas about partial derivatives.

As shown in Figure 1, we have extended the standard Coq library with equivalent definitions that are easier to use and with some automations. The developed library can be downloaded and browsed at

<http://coquelicot.saclay.inria.fr/results.html>



**Fig. 1.** Sketch of our library: equivalent definitions, additional lemmas, and an automatic tactic, without introducing new axioms

To validate this approach, we have applied it to an example coming from a numerical analysis program. We had previously proved the correctness of a program solving the wave equation, but we left out the proofs about the existence and regularity of a solution to the partial differential equation.

Section 2 presents the existing formalizations of analysis in proof assistants, and especially in Coq. Section 3 presents the power of the underlying logic of Coq, especially the changes when considering the axiomatic of real numbers. Section 4 presents our design choices for the derivative and the integral. Section 5 presents our application about the wave equation and its required results.

## 2 State of the Art

We will present the notions of differentiability and integrability as they are defined in various proof assistant such as Coq<sup>[1]</sup>, the Coq constructive library C-CoRN<sup>[2]</sup>, HOL Light<sup>[3]</sup>, Isabelle/HOL<sup>[4]</sup>, PVS<sup>[5]</sup>, Mizar<sup>[6]</sup>, and ACL2(r)<sup>[7]</sup>. We will first present the definitions of the differentiability and integrability predicates, and then describe the design choices for the derivative and integral functions.

### 2.1 Differentiability and Integrability

The choices made here are to adopt one of the common mathematical definitions. For differentiability, Coq and PVS use  $\varepsilon$ - $\delta$ -definition based on the Landau definition directly or through the limit definition:

$$\exists \ell \in \mathbb{R}, \forall \varepsilon > 0, \exists \delta > 0, \forall h \in \mathbb{R}, (h \neq 0 \wedge |h| < \delta) \Rightarrow \left| \frac{f(x+h) - f(x)}{h} - \ell \right| < \varepsilon$$

In HOL Light (and Isabelle/HOL which has inherited that formalization of analysis), the Newton's difference quotient is also used, but the limit is a more generic notion as it is defined in a topological space based on a field [12]. Another approach is implemented in additional libraries: the Frechet derivative in a real normed vector space [13]. In C-CoRN, a constructive formalization of real numbers for Coq, the previous definition is modified to get a uniform differentiability [5] on a closed interval  $[a; b]$ , *i.e.* there is a single  $\delta$  for all  $x \in [a; b]$ . In Mizar, differentiability is defined for multi-variable functions [17] as usual by the existence of a linear function  $L$  such that  $f(x+h) - f(x) - L(h) = o(h)$ .

To define Riemann integrability, Coq defines the integral on step functions and then uses the traditional  $\varepsilon$ - $\delta$ -definition:

$$\forall \varepsilon > 0, \text{ there are two step functions } \varphi, \psi : [a; b] \rightarrow \mathbb{R}, \text{ such that} \\ (\forall t \in [a; b], |f(t) - \varphi(t)| \leq \psi(t)) \wedge \int \psi < \varepsilon$$

The value of the integral is then defined as the limit of  $\int \varphi$  when  $\varepsilon \rightarrow 0$ .

PVS in [4] and Mizar in [7] define Riemann integrability as the convergence of Riemann sums. Both definitions are mathematically equivalent. A difference is that the integral value is explicitly given by this second definition while Coq's definition only provides approximations of Riemann integral.

C-CoRN uses a third equivalent definition of Riemann integrability: the convergence of Darboux sequences. As with Riemann sums, the integral value is directly obtained from the definition.

<sup>1</sup> <http://coq.inria.fr/stdlib/index.html>

<sup>2</sup> <http://corn.cs.ru.nl/>

<sup>3</sup> <http://www.cl.cam.ac.uk/~jrh13/hol-light/>

<sup>4</sup> <http://isabelle.in.tum.de/dist/library/HOL/index.html>

<sup>5</sup> <http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/>

<sup>6</sup> <http://www.mizar.org/>

<sup>7</sup> <http://www.cs.utexas.edu/users/moore/acl2/v5-0/>

HOL Light does not define Riemann integrals, but both gauge and Lebesgue integrals, which are more general but less intuitive. As with derivatives, all the notions are defined for multivariate functions.

In a former library about reals in Isabelle/HOL [8], and the current library in ACL2(r) [9], differentiability was defined from non-standard analysis where the formal notion of “infinitely close”, *i.e.* difference is less than all positive real numbers, replaces the usual notion of “arbitrarily close” corresponding to a common formula stating that  $\forall \varepsilon > 0, \dots$ , the difference may be made smaller than  $\varepsilon$ .

## 2.2 Derivative and Integral

In pen-and-paper mathematics, we understand  $f'(x)$  as the function  $f$  is differentiable at  $x$  and  $f'(x)$  is its derivative value. But the corresponding definitions and their uses in formal proof assistants are not straightforward and differ, among others as the underlying logic is different.

PVS is the closest to the mathematical point of view by using Type Correctness Conditions (TCCs). A user may write a statement containing occurrences of  $f'(x)$  without justification and PVS generates additional goals to prove that  $f$  is actually differentiable in  $x$ . PVS tries to infer automatically these additional goals from the context. The same approach is used in ACL2(r) [10].

HOL Light uses its Hilbert’s epsilon applied to the differentiability property. The user can then write  $f'(x)$  without proof, but has to prove that  $f$  is differentiable at  $x$  before using differentiation rules.

In Coq, the derivative is a partial function taking explicitly a proof term of differentiability. As a derivative cannot be defined or used without this proof term, this is cumbersome to use. One of the goals of the MathClasses project [18] is to provide assistance work with these proof terms by trying to infer them.

## 3 Logical Foundations

Rather than introducing our own set of axioms, we have based our formalization on the axioms from the Coq standard library on real numbers. This ensures that our automated proof tools are usable by works based on the standard library, that some results from the standard library could be reused, and that our work is consistent (assuming the standard library is). We have also taken great care to not introduce any other axioms, especially not the excluded middle, which is neither a native concept in Coq logic, nor a consequence of Coq axioms on real numbers.

### 3.1 An Overview of Coq’s Logic

The formal system of Coq is an intuitionistic logic called the Calculus of Inductive Constructions. A salient point is that, whenever one wants to prove a property of the form  $\exists x, P(x)$ , one has to actually build a witness  $x$  such that  $P(x)$  holds. This is different from classical logic where one could have simply proved  $\neg \forall x, \neg P(x)$  and be done.

Another peculiarity of Coq’s logic is related to its type hierarchy. Logical formulas have type **Prop** while values and functions have their types in **Type**. The point of interest is that **Prop** is *non-informative*, that is, one can only use the witness of an existential property  $\exists x, P(x)$  inside the proof of logical formulas. A witness can never be used inside a logical formula itself, or more importantly to define a value or a function. For instance, being able to prove a formula  $\forall x : X, \exists y : Y, P(x, y)$  does not provide a function  $f : X \rightarrow Y$  such that  $\forall x, P(x, f(x))$ . Note that strengthening the property so that  $y$  exists and is unique does not help either.

The traditional way in Coq to circumvent this issue is in the use of *specification* types. They are existential types denoted  $\{x : X \mid P(x)\}$  and they contain dependent pairs  $(x, p)$  such that  $x$  is an element of type  $X$  and  $p$  is a proof of the logical formula  $P(x)$ . The upside is that witnesses are readily available for use inside values and functions. The downside is that these types do not live in **Prop** and are therefore less natural to manipulate.

For instance, consider the predicate `derivable_pt` from the standard library that states that a function  $f$  is differentiable at point  $x$ . It is in fact a notation for the specification type that associates a value  $\ell$  with the proof that  $\ell$  is the limit of the slope function of  $f$  at point  $x$ . As a consequence, knowing that a function is differentiable gives access to the value of its derivative. But it also means that trying to express that both  $f$  and  $g$  are functions that are differentiable as `derivable_pt(f, x)  $\wedge$  derivable_pt(g, x)` will be rejected by Coq. Indeed, this formula is ill-typed since neither members are logical formulas. A coercion from specification types to formulas in **Prop** would avoid this issue as long as one only needs the differentiability property and not the derivative of a function.

In the same way that one can extract a witness from the formula  $\exists x, P(x)$  only when performing a proof, the information about the disjunction  $P \vee Q$  cannot be used to make a choice inside a value or a function. Again, types outside **Prop** have been introduced to offer this possibility; they are denoted  $\{P\} + \{Q\}$  in Coq and their values can be used to select the branch of an if-then-else.

To end this overview, one should mention that subsets of a type  $T$  are usually represented by predicates, that is, functions of type  $T \rightarrow \mathbf{Prop}$ . As a consequence, we will indifferently note the containment property by the logical formulas  $x \in S$  or  $S(x)$ .

### 3.2 Coq’s Standard Axioms for Real Numbers

The formalization of real numbers from the standard library is axiomatic rather than definitional. Instead of building reals as Cauchy sequences or Dedekind cuts of rational numbers and proving their properties, Coq developers have chosen to assume the existence of a set with the usual properties of the real line. In other words, the standard library states that there is a set  $\mathbf{R}$ , some arithmetic operators  $-$ ,  $+$ ,  $\times$ ,  $\square^{-1}$ , and a comparison operator  $<$ , that have the properties of an ordered field. Except perhaps for the choice for the domain of  $\square^{-1}$ , the previous axioms are not controversial.

Below are the three axioms that state that  $\mathbb{R}$  is Archimedean, closed under the supremum bound, and its order is decidable:

- **archimed**: There is a function  $\text{up} : \mathbb{R} \rightarrow \mathbb{Z}$  such that  $\forall x \in \mathbb{R}, x < \text{up}(x) \leq x+1$ .
- **completeness**: As long as one can prove that a subset  $E$  of  $\mathbb{R}$  is not empty ( $\exists x, E(x)$ ) and is bounded ( $\exists M, \forall x, E(x) \Rightarrow x \leq M$ ), one gets a value of type  $\{y : \mathbb{R} \mid y \text{ is an upper bound of } E \text{ and it is the least one}\}$ .
- **total\_order\_T**: Given two real numbers  $x$  and  $y$ , there is a value of type  $\{x < y\} + \{x = y\} + \{x > y\}$ .

While all three axioms could have been defined as logical formulas in **Prop**, they were not. In other words, it is equivalent to having three functions that can compute the integer part of a real number, the supremum of a bounded subset of  $\mathbb{R}$ , and the order of two numbers. Notice that excluded-middle is not one of the axioms of Coq’s standard real numbers. While the standard library sometimes imports this axiom, the CoqTail project<sup>8</sup> has shown that it was often unneeded. So we restrict our use of classical reasoning to the proof of goals that are the negation of a logical formula.

Notice also that the **completeness** axiom has an intuitionistic rather than classical feel to it. Indeed, the axiom is written in such a way that it cannot produce a proof of  $E(x)$  for some real  $x$  (especially not the real that is the supremum of  $E$ ), since properties of the form  $E(x)$  always appear as premises in the axiom. At best, one can derive the following property which protects the existential quantifier behind a double negation:

$$\forall \varepsilon > 0, \neg\neg\exists x, y - \varepsilon \leq x \leq y \wedge E(x).$$

### 3.3 Limited Principle of Omniscience

Let  $P$  be a decidable predicate on natural numbers. The limited principle of omniscience (LPO) states that one can decide whether the property never holds, and if it does, return  $n$  such that  $P(n)$  holds. In Coq syntax, the principle is stated  $\{n \mid P(n)\} + \{\forall n, \neg P(n)\}$ . It cannot be derived without axioms in Coq, as it would require the ability to test all the values of  $n$  at once. Thanks to the axioms on real numbers, it becomes possible.

The way we have proved the LPO is as follows. Since  $P$  is decidable, we can build a function  $f(n)$  that returns  $1/(n+1)$  if  $P(n)$  holds and 0 otherwise. Let us consider the subset of real numbers  $\{f(n) \mid n \in \mathbb{N}\}$ . It is nonempty and bounded by 1, thus it has a supremum (**completeness**). This supremum can be tested against 0 (**total\_order\_T**). If it is zero, we deduce  $\forall n, \neg P(n)$ . Otherwise we compute its discrete inverse (**archimed**) which will act as a witness for building a value of type  $\{n \mid P(n)\}$ .

This proof of the LPO, inspired by the CoqTail project, is an improvement over some previous work that depended on the **not\_all\_ex\_not** consequence of the excluded-middle axiom [15].

<sup>8</sup> <http://coqtail.sourceforge.net/>

### 3.4 Bounds and Limits

Now that we have proved this principle, we can use it to decide whether a subset  $E$  of real numbers is bounded, which is a precondition for computing its supremum. First, let us consider the family of subsets  $E_n = \{0\} \cup (E \cap (-\infty; n])$ . They are nonempty and bounded, so they have a supremum  $s_n$ . As a consequence, deciding whether  $E$  is bounded and computing an upper bound is a matter of applying the LPO to decide the following alternative (Lemma `Rbar_ub_dec` of our development):

$$\{M \mid \forall n, s_n \leq M\} + \{\forall M, \neg(\forall n, s_n \leq M)\}.$$

The proof requires that  $\forall n, s_n \leq M$  is decidable, which is just a consequence of the LPO applied to the decidable predicate  $n \mapsto M < s_n$ .

Let us define the complete lattice `Rbar` =  $\mathbb{R} \cup \{-\infty, +\infty\}$ , inspired by [14]. Since we are able to decide whether a set is bounded, we can define supremum and infimum functions for nonempty subset of `Rbar`.<sup>9</sup> Let us consider a sequence  $(u_n)_{n \in \mathbb{N}}$  of elements of  $\mathbb{R}$  (or `Rbar`). The set of its values is nonempty, thus we can define its superior and inferior limits:

$$\limsup (u_n)_{n \in \mathbb{N}} = \inf_{m \in \mathbb{N}} (\sup_{n \in \mathbb{N}} u_{m+n}) \quad \liminf (u_n)_{n \in \mathbb{N}} = \sup_{m \in \mathbb{N}} (\inf_{n \in \mathbb{N}} u_{m+n})$$

At this point, we arbitrarily define a function `Lim_seq` from sequences to real numbers as  $(\limsup(u_n) + \liminf(u_n))/2$ , or 0 in case of infinities. If  $(u_n)_n$  is a converging sequence, `Lim_seq`  $(u_n)_{n \in \mathbb{N}}$  is the actual limit of the sequence, since inferior, superior, and plain limits are then equal. Note that this also gives us a way to decide whether an arbitrary sequence is converging: we just have to compare its inferior and superior limits. With this definition, we also get “limits value” for non-convergent sequences: `Lim_seq`  $(-1)^n = (1 + (-1))/2 = 0$ .

There were simpler possibilities for defining `Lim_seq`, but this one offers the equality `Lim_seq`  $(\alpha \cdot u_n) = \alpha \cdot \text{Lim\_seq}(u_n)$  for any real  $\alpha$  without requiring  $(u_n)_n$  to converge.

We now have an operator able to compute the limit of any converging sequence and otherwise return an undefined value (as far as the user is concerned). We can similarly define the limit of a function  $f$  in a point  $x$  by

$$\text{Lim}(f, x) = \text{Lim\_seq} \left( f \left( x + \frac{1}{n} \right) \right)_{n \in \mathbb{N}}.$$

Again, if the function has a limit at this point, in sense of the usual  $\varepsilon - \delta$  definition for pointed limit, operator `Lim` returns it, since  $\lim_{u \rightarrow x} f(u)$  is then equal to  $\lim_{n \rightarrow \infty} f(u_n)$  for any sequence such that  $\lim(u_n) = x$  and  $\forall n \in \mathbb{N}, u_n \neq x$ . Otherwise it does return a value, but a meaningless one. We did not gain anything significant by defining the limit with more complicated sequences such as  $(f(x - 1/n) + f(x + 1/n))/2$ .

<sup>9</sup> Supremum and infimum of a subset  $P$  are extended with  $\sup \emptyset = -\infty$  and  $\inf \emptyset = +\infty$ , when one can prove  $P \neq \emptyset \Rightarrow \exists x, P(x)$ . This generalization is not needed here.



### 3.5 Compactness

Limits are the basis for doing real analysis. Another important tool is the property of compactness, which has numerous applications in traditional mathematics. For instance, a function continuous on a compact set is uniformly continuous. Unfortunately, the compactness property is inherently classical, up to the point that constructive mathematics tend to redefine continuity so that it actually means uniform continuity in order to avoid compactness [6]. Our goal is to stay as close as possible to traditional analysis, so dropping compactness is not a solution.

One of the definitions of a compact set is a set such that, from any cover with open sets, one can extract a finite subcover. Yet in most of the proofs we are interested in, we do not need the whole power of this property. Indeed, the extracted sets are useless, only their minimum diameter matters. Moreover, the finiteness property is only useful so that this minimum is nonzero. As a consequence, we can substitute to the traditional definition a property of interval  $[a, b]$  related to Cousin covers and gauge functions:

$$\forall \delta : \mathbb{R} \rightarrow \mathbb{R}^+, \{ \delta' : \mathbb{R}^+ \mid \forall x \in [a, b], \neg \exists t \in [a, b], |x - t| < \delta(t) \wedge \delta' \leq \delta(t) \}.$$

The proof of this lemma (`compactness_value`) requires us to construct a value  $\delta'$  that satisfies the property. It is defined by the formula

$$\delta' = \frac{1}{2} \sup \{ d \mid d \leq 1 \wedge \forall x \in [a, b], \exists t \in [a, b], |x - t| < \delta(t) \wedge d \leq \delta(t) \}.$$

First, we prove that this supremum is not equal to zero. This is the same idea as with most cover-based proofs: consider the widest interval  $[a, b']$  such that the property  $\exists d > 0, \forall x \in [a, b'], \exists t \in [a, b], |x - t| < \delta(t) \wedge d \leq \delta(t)$  holds. The bound  $b'$  is defined as a supremum value and we prove that, if  $b' < b$ , there is a contradiction. Then we prove that it still holds for the whole interval  $[a, b]$ , so  $\delta'$  is positive.

Second, we prove that  $\delta'$  satisfies the original property. This is immediate, but only because the property contains a double negation. Indeed, as mentioned before, the completeness axiom never provides us with proofs that the property holds for values smaller than the supremum. So the best we can prove is that there would be contraction if  $\delta'$  did not satisfy the property. In practice, this double negation does not matter because we always use the compactness property to exhibit contradictions.

Finally, we have proved the compactness property not just for segments of the real line but for any  $n$ -orthotope (Cartesian product of  $n$  segments).

## 4 Derivatives and Integrals

### 4.1 Derivative

From the previous definition of a limit function in Section 3.4, we defined a total derivative function:

$$\text{Derive}(f, x) = \lim_{\substack{h \rightarrow 0 \\ h \neq 0}} \frac{f(x + h) - f(x)}{h}$$

and we proved that, if the derivative number  $f'(x)$  exists, it is equal to our derivative number using lemmas proved in the standard library.

This definition allows to write properties about derivative without any proof term. For example to write “the function  $f$  is differentiable on the domain  $D$  and  $f'$  satisfies the property  $P$  on this domain” in Coq, we previously had to write

```
Lemma pr : forall x, D x → derivable_pt f x.
Goal forall x (Dx : D x), P (derive_pt f x (pr x Dx)).
```

using a dependent pair. With our approach, we can write the same statement as

```
Goal forall x, D x → ex_derive f x.
Goal forall x, D x → P (Derive f x).
```

and then we can prove separately differentiability and the property  $P$ . Moreover, as our derivative is a limit, some properties such as  $(\alpha \cdot f)' = \alpha \cdot f'$  do not require the differentiability of  $f$ . Therefore, the proof burden is lightened, as theorems have less preconditions. Such a property could also be obtained for a derivative defined on top of Hilbert’s epsilon, but not without introducing a cumbersome definition based on conditionals. Our approach also makes it possible to express more easily the  $n$ -th derivative:

$$\begin{cases} \text{Derive}_n(f, 0, x) &= f(x) \\ \text{Derive}_n(f, n + 1, x) &= \text{Derive}(x \mapsto \text{Derive}_n(f, n, x), x) \end{cases}$$

and we express the  $n$ -th differentiability as

```
forall i : nat, (i < n)%nat → ex_Derive (Derive_n f i) x
```

## 4.2 Riemann Integral and Riemann Sums

We define our integral `RInt` as the limit of

$$\text{RInt\_val}(f, a, b, n) = \frac{b - a}{n + 1} \sum_{k=0}^n f \left( \frac{x_k^n + x_{k+1}^n}{2} \right)$$

where  $x_k^n = a + k \cdot (b - a)/(n + 1)$ . As above, this definition allows to prove  $\forall \alpha \in \mathbb{R}, \text{RInt}(\alpha \cdot f, a, b) = \alpha \cdot \text{RInt}(f, a, b)$  without hypotheses on  $f$ .

The Riemann integrability in the standard library is based on step functions. Unfortunately, step functions from the standard library are hard to use: a step function is built from a function  $f$  and two lists  $lx$  and  $ly$  that must satisfy five conditions. These conditions are difficult both to prove and to use. For example, the last one states that  $\forall i, \forall x \in (lx_i; lx_{i+1}), f(x) = ly_i$ . This is impractical as it does not provide any information about the values  $f(lx_i)$ .

We chose to define new step functions based on `Ssreflect` sequences [\[11\]](#):

```
Record SF_seq {T : Type} := mkSF_seq {SF_h : R ; SF_t : seq (R * T)}.
```

Using `Ssreflect` libraries, our step functions were easier to define and use. We define our step functions with a generic type `T` to use it in the same way with `T := R` or `T := Rbar`. For example, to define the step function needed for `RInt_val(f, a, b, n)`, we use

**Definition** `SF_val_seq` (`f : R → R`) (`a b : R`) (`n : nat`) : `SF_seq` :=  
`SF_seq_f2` (`fun x y => f ((x+y)/2)`) (`RInt_part a b n`) 0.

where `RInt_part a b n` is the partition used in this proof and `SF_seq_f2` builds the needed step function.

Moreover, as the standard library does not provide a global relation between Riemann integrability and Riemann integral, we chose to prove the equivalence between this definition and the following convergence of Riemann sums:

$$\exists If \in \mathbb{R}, \forall \varepsilon > 0, \exists \delta > 0, \forall (\sigma, \xi), \max_{0 \leq k \leq n} |\sigma_{k+1} - \sigma_k| < \delta \Rightarrow |S(f, \sigma, \xi) - If| < \varepsilon$$

where  $S(f, \sigma, \xi) = \sum_{k=0}^n f(\xi_k) \cdot (\sigma_{k+1} - \sigma_k)$  is a Riemann sum,  $n$  is the length of  $\xi$ ,  $n+1$  the length of  $\sigma$ ,  $\sigma_0 = a$ ,  $\sigma_{n+1} = b$  and  $(\sigma, \xi)$  is a pointed subdivision, i.e.  $\forall k \leq n, \sigma_k \leq \xi_k \leq \sigma_{k+1}$ . This is the same approach as in PVS and Mizar.

Thanks to this new definition, we can express integrability with the same structure as convergence and differentiability: a function gives the value of the Riemann integral, a predicate states the property of Riemann integrability, and `is_RInt` (`f, a, b, If`) states that `If` is the Riemann integral of `f` between `a` and `b`.

We can note that our value is the limit of a sequence of specific Riemann sums  $S(f, \sigma_n, \xi_n)$  such that  $\forall i \in \llbracket 0; n \rrbracket, \sigma_{i+1} - \sigma_i = (b - a)/(n + 1)$ . The correctness is then immediate for our new definition of Riemann integrability.

To ensure the compatibility with the standard library of Coq, we proved the equivalence between both definitions, so that we can take advantage of existing lemmas.

## 5 Application

### 5.1 Case Study and d'Alembert's Formula

Our main application is part of a project aiming at proving numerical analysis programs. The case study was a C program that implements a numerical scheme for the resolution of the one-dimensional acoustic wave equation. This corresponds to the oscillation of an attached rope where  $c$  is the constant propagation velocity, which depends on the section and density of the string. More precisely, we consider the following initial-boundary value problem: we have the initial values  $u_0$  and  $u_1$ , a source term  $s$  and we want to compute an approximation of the exact solution  $u$  of

$$\begin{aligned} \forall t \geq 0, \forall x \in [x_{\min}, x_{\max}], \quad & \frac{\partial^2 u}{\partial t^2}(x, t) - c^2 \frac{\partial^2 u}{\partial x^2}(x, t) = f(x, t), \\ \forall x \in [x_{\min}, x_{\max}], \quad & \frac{\partial u}{\partial t}(x, 0) = u_1(x), \\ \forall x \in [x_{\min}, x_{\max}], \quad & u(x, 0) = u_0(x), \\ & \forall t \geq 0, u(x_{\min}, t) = u(x_{\max}, t) = 0 \end{aligned}$$

To actually compute an approximation of the solution  $u$ , we chose the second order centered finite difference scheme, also known as three-point scheme. The size of the grid is  $(\Delta x, \Delta t)$  and the value  $u_j^k \approx u(j\Delta x, k\Delta t)$  is given by

$$\frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} - c^2 \frac{u_{j+1}^{k-1} - 2u_j^{k-1} + u_{j-1}^{k-1}}{\Delta x^2} = s_j^{k-1}$$

and similar formulas that depend on  $u_0$  and  $u_1$  for the initializations  $u_j^0$  and  $u_j^1$ .

We proved that rounding errors do not endanger the results of the numerical scheme [1]. We also formalized in Coq the numerical scheme and proved its convergence, for an infinite rope [2]. In that work, the differentiation operator was an arbitrary function. We did not define it nor did we assume any of its properties. We only used the fact that  $u$  is a solution of the partial differential equation expressed using this operator. This fully corresponds to the way mathematical proofs are done: we put  $f'(x)$  or  $\frac{\partial^2 u}{\partial t^2}(x, t)$  and not  $\text{diff}(f, x, H)$  where  $H$  is a proof that  $f$  is derivable in  $x$  or  $\text{diff}(z \rightarrow \text{diff}(y \rightarrow u(x, y), z, H_1), t, H_2)$  where  $H_1$  and  $H_2$  are adequate proof terms. We also needed the regularity of this solution: it is supposed to be near its Taylor expansion with the usual mathematical bounds (see below).

Later, we proved the full C program for a finite rope [3]. As we needed to precisely specify what the program was supposed to compute, we defined each derivative as the limit of  $\frac{f(x+h)-f(x)}{h}$  when  $h$  goes to zero. We used the Frama-C platform with the Jessie plugin and the specification of the C program is described in C comments called annotations. As the language of these annotations is first-order logic, we could not define in our specifications a differentiation operator, but had to define each of the four derivatives as a limit (with a  $\forall \epsilon, \exists \delta \dots$  formula). This fully specifies the derivatives but was impractical and difficult to read. Yet, this meant an equivalence between our previous work and real derivatives. But as our previous work required a differentiation operator, we had to actually provide it. The chosen solution was to define it as a parameter and add an axiom stating that, if the function is differentiable, the result of this operator is the expected derivative. This axiom, similar to a Hilbert  $\epsilon$  operator, was not satisfactory. Thanks to the formalization presented in this paper, we can now create this operator as a function and get rid of that axiom.

The other axioms needed by this development are the following ones: the existence of a solution to the partial differential equation and its regularity.

About the existence, the mathematical proof is simpler than expected as this equation has an analytical solution. More precisely, the following d'Alembert's formula

$$u(x, t) = \underbrace{\frac{1}{2} (u_0(x + ct) + u_0(x - ct))}_{\alpha(x, t)} + \underbrace{\frac{1}{2c} \int_{x-ct}^{x+ct} u_1(\xi) d\xi}_{\beta(x, t)} + \underbrace{\frac{1}{2c} \int_0^t \int_{x-c(t-\tau)}^{x+c(t-\tau)} f(\xi, \tau) d\xi d\tau}_{\gamma(x, t)}$$

defines a function that is solution to the previous partial differential equation. We define  $\alpha, \beta$ , and  $\gamma$ , as parts of this formula that will be used below. Just note that they are of increasing difficulty to handle and derive.

## 5.2 Taylor expansions

The regularity of the solution  $u$  is the base of the convergence of the chosen numerical scheme. In the scheme statement, since the grid sizes are small, we can recognize discrete derivatives:

$$\frac{u_j^k - u_j^{k-1}}{\Delta t} \approx \frac{\partial u}{\partial t}(j\Delta x, k\Delta t) \quad \frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} \approx \frac{\partial^2 u}{\partial t^2}(j\Delta x, k\Delta t)$$

and similarly for space derivatives. The discrete equation is the exact discrete analog of the continuous wave equation.

Our definition of the Taylor polynomial is the usual one:

$$\text{TP}_n(f, x, t) = \sum_{p=0}^n \frac{1}{p!} \left( \sum_{m=0}^p \binom{p}{m} \cdot \frac{\partial^p f}{\partial x^m \partial t^{p-m}}(x, t) \cdot \Delta x^m \cdot \Delta t^{p-m} \right)$$

For the main iteration, we need to guarantee that the difference between the function and its order-4 Taylor polynomial is proportional to  $(\sqrt{\Delta x^2 + \Delta t^2})^4$ . For the initializations, we also need this property at level 3.

We first proved the common Taylor-Lagrange theorem and extended it to its two-dimensional version we really needed here. We had to prove the Schwarz theorem to be able to switch derivatives in space and time. Note that the hypotheses required to make this switch possible are strong (existence and continuity of both second-order derivatives).

## 5.3 Automation

As explained, the two unproved properties from the original development are that the solution exists and is sufficiently regular. Existence has already been tackled thanks to a reflexive Coq tactic for proving differentiability [16]. The tactic was limited: it could handle expressions with one variable only. As a consequence, while it could automatically perform differentiability proofs on the  $\alpha$  and  $\beta$  parts of d'Alembert's formula, human intervention was needed for differentiating under the integral sign of  $\gamma$ . The reason for this shortcoming was the need for proof terms in derivatives and integrals. For instance, the term  $\int_{a(x)}^{b(x)} \frac{\partial f}{\partial x}(x, t) dt$  contains a proof that  $\frac{\partial f}{\partial x}(x, t)$  can be integrated for  $t$  between  $a(x)$  and  $b(x)$ , while the term  $\frac{\partial f}{\partial x}(x, t)$  itself contains a proof that  $f$  has a first derivative at any point  $(x, t)$  of a domain that depends on  $x$ . This nesting of values and proof terms ended up being out of reach of our tactic.

For the existence, we only had to consider four partial derivatives of  $\gamma$ . So, despite the absence of automation, we succeeded in formally proving it in Coq. For this work, however, we wanted to prove not only the existence but also the regularity, which means manipulating tens of partial derivatives of  $\gamma$ . This makes it out of reach of a non-automated proof. So we have improved the original tactic now that we have got rid of proof terms in values. The new tactic `auto_derive` still produces side conditions, but the values no longer depend on their proofs,

which means the tactic is now able to differentiate below the integral sign. As the former one, this tactic is programmed in `Ltac`.

The tactic is meant to help proving statements of the form

```
derivable_pt_lim f x l
```

that is,  $f$  has a derivative at point  $x$  and it is equal to  $l$ . A variant of the tactic is able to tackle goal where  $l$  is not yet known. The tactic first performs a reification of the function  $f$  into an inductive object describing the expression. Variables are encoded using De Bruijn's indexes. For instance, the inductive object for

$$y \mapsto \int_0^{2y} (g(y) + z) dz$$

is

```
(Int                                     (* integral *)
  (Binary Eplus                          (* addition *)
    (AppExt 1 g [:: Var 1])              (* application: g(y) *)
    (Var 0))                             (* z *)
  (Cst 0)                                 (* lower bound of integral: 0 *)
  (Binary Emult (Cst 2) (Var 0)))        (* upper bound: 2y *)
```

Operator `D` is then applied to this inductive object. This is a recursive function that differentiates an expression and generates side conditions. For instance, given an object representing  $x \mapsto 2 \cdot f(x)$ , it produces an object representing  $x \mapsto 2 \cdot f'(x)$  and a side condition that  $f$  can be differentiated at the considered point. Lemma `D_correct` states that the generated object is the actual derivative when the side conditions hold. The tactic simply applies this lemma to the current goal, thus solving it, assuming the user can prove the side conditions.

Consider the following script that proves that  $\frac{\partial^2 \alpha}{\partial x^2}$  exists and is equal to  $\alpha_{20}$ .

```
Definition alpha x t := 1/2 * (u0 (x + c*t) + u0 (x - c*t)).
Definition alpha20 x t :=
  1/2 * (Derive_n u0 2 (x + c*t) + Derive_n u0 2 (x - c*t)).
```

```
Lemma alpha_20_lim : forall x t,
  is_derive_n (fun u => alpha u t) 2 x (alpha20 x t).
```

**Proof.**

```
intros x t. unfold alpha.
auto_derive_2.
```

The `auto_derive_2` is just an ad-hoc tactic developed for this example. It simply applies `auto_derive` twice in a row, so as to prove properties on iterated derivatives. After executing the tactic, the user is left with several goals to prove. They state that function  $u_0$  can be differentiated with respect to the first variable at points  $(v + c \cdot t)$  and  $(v - c \cdot t)$  for an arbitrary real  $v$  around  $x$ . They also state that the first partial derivative of  $u_0$  can be differentiated with respect to the first variable at points  $(x + c \cdot t)$  and  $(x - c \cdot t)$ . Finally, the last goal the user has to prove is the equality between  $\alpha_{20}(x, t)$  and the expression obtained by automatic differentiation, which is straightforward.

## 6 Conclusion

We have presented a Coq development for real analysis that aims at being closer to the traditional way of handling analysis theorems in pen-and-paper proofs. The main idea we have followed is to replace all the partial operators with total operators, so that the user no longer has to manipulate dependent types. Once all the theorems and especially rewriting rules have non-dependent hypotheses, they become much easier to apply, since reasoning is back to being backward: from the goal to the hypotheses.

The standard libraries of HOL Light and Isabelle/HOL also provide such total operators for derivatives and integrals. The main difference with our work is that they are defined thanks to Hilbert  $\varepsilon$  operator, which is not available in Coq. Instead, we have provided algorithms for these operators. They are not the kind of algorithms that one would find in traditional constructive mathematics, since our model of computation is a bit unusual. It has a decidable order, as in the Real RAM model, but it also has a supremum operator and an integer part.

To obtain this model, we have not added any axiom, we have just reused the axiomatization of real numbers from the Coq standard library. We have also taken great care to never use the axiom of excluded middle, contrarily to what is done in the standard library. Unfortunately, when looking at the assumptions of some of the theorems of our library, there might be occurrences of this axiom. They leak from the standard library through the equivalence lemmas between our definition of integral and the standard one. Work is under way to remove these uses from the standard library and therefore get a formalization that no longer relies on excluded middle.

Our development of more than 500 lemmas provides total operators for limits, derivatives, and integrals, and equivalence lemmas between our constructive definitions and the partial operators from the standard library. We have also extended the theory of real analysis further than what is available in the standard library: iterated partial derivatives, parametric integrals, and so on.

We have applied our formalization to filling the holes in the formal verification of a numerical program: the three-point scheme for solving the one-dimensional wave equation. The original formalization set as an axiom the existence of total operators for partial derivatives; the verification would never have succeeded if it had had to cope with the dependent types needed for expressing fourth derivatives. The work presented in this paper fills this gap.

The original formalization was also assuming the existence and the regularity of a solution to the partial differential equations. When starting from d'Alembert formula, the formal proof of these properties is mostly mechanical. One just has to differentiate the formula as many times as needed (up to 20 times for order-4 regularity). For that purpose, we have also developed a reflexive Coq tactic that is able to perform such repetitive tasks. The strength of our tactic is that it is able to differentiate under the integral sign. As far as we know, no similar strategy has been developed for other provers.

## Future Works

For this work, we chose Riemann's definition of integral. Our only motivation for this choice was that it is the integral provided by Coq's standard library and we wanted to check that we were not less expressive than the standard library. If not for this constraint, we would have chosen a different definition, *e.g.* Lebesgue integral. Indeed, compared to other definitions of integral, Riemann integral does not have much positive points, except for its prestigious name. Lebesgue integral would have been easier for us to define and to manipulate, since it is almost a simple supremum.

In fact, we could presumably go further than Lebesgue integral and define gauge integral. Indeed, our work on compactness has shown that our limited framework (no general excluded middle) was still sufficient to manipulate gauge functions and extract finite subcovers. This is the main property needed to prove Cousin's theorem and therefore to define this integral. It has two main positive points. First, it is no more complicated than Riemann integral. Second, the second fundamental theorem of calculus can now be expressed without any precondition: each differentiable function is the integral of its derivative. This makes its usage in formal proofs straightforward.

For now, we have only defined limits, derivatives, and integrals, as total operators. Our goal is to extend this paradigm to other common operators, *e.g.* power series and reciprocal, so as to provide all the basic blocks of analysis. We also intend to extend our automated tools beyond just differentiation. An obvious extension is integration, but also automatic proofs of integrability and continuity. Indeed, differentiating under the integral sign tends to generate numerous side conditions about these properties and they would greatly benefit from being automatically discharged by the prover during the symbolic computations.

## References

1. Boldo, S.: Floats and Ropes: A Case Study for Formal Numerical Program Verification. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikolettseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 91–102. Springer, Heidelberg (2009)
2. Boldo, S., Clément, F., Filliâtre, J.-C., Mayero, M., Melquiond, G., Weis, P.: Formal Proof of a Wave Equation Resolution Scheme: The Method Error. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 147–162. Springer, Heidelberg (2010)
3. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave Equation Numerical Resolution: a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning* (accepted for publication on May 20, 2012), <http://hal.inria.fr/hal-00649240>
4. Butler, R.W.: Formalization of the integral calculus in the PVS theorem prover. *Journal of Formalized Reasoning* 2(1), 1–26 (2009)
5. Cruz-Filipe, L.: Constructive Real Analysis: a Type-Theoretical Formalization and Applications. Ph.D. thesis, University of Nijmegen (April 2004)



6. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-CoRN: the constructive Coq repository at Nijmegen. In: 3th International Conference on Mathematical Knowledge Management (MKM), Bialowieza, Poland, pp. 88–103 (2004)
7. Endou, N., Kornilowicz, A.: The definition of the Riemann definite integral and some related lemmas. *Journal of Formalized Mathematics* 8(1), 93–102 (1999)
8. Fleuriot, J.: On the Mechanization of Real Analysis in Isabelle/HOL. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 145–161. Springer, Heidelberg (2000)
9. Gamboa, R.: Continuity and differentiability in ACL2. In: *Computer-Aided Reasoning: ACL2 Case Studies*, ch. 18. Kluwer Academic Publisher (2000)
10. Gamboa, R., Kaufmann, M.: Non-standard analysis in ACL2. *Journal of Automated Reasoning* 27(4), 323–351 (2001)
11. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Tech. Rep. RR-6455, INRIA (2008), <http://hal.inria.fr/inria-00258384>
12. Harrison, J.: *Theorem Proving with the Real Numbers*. Springer (1998)
13. Harrison, J.V.: A HOL Theory of Euclidean Space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
14. Hölzl, J., Heller, A.: Three Chapters of Measure Theory in Isabelle/HOL. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 135–151. Springer, Heidelberg (2011)
15. Kaliszyk, C., O’Connor, R.: Computing with classical real numbers. *Journal of Formalized Reasoning* 2(1), 27–39 (2009)
16. Lelay, C., Melquiond, G.: Différentiabilité et intégrabilité en Coq. Application à la formule de d’Alembert. In: 23èmes Journées Francophones des Langages Applicatifs, Carnac, France, pp. 119–133 (2012)
17. Raczkowski, K., Sadowski, P.: Real function differentiability. *Journal of Formalized Mathematics* 1(4), 797–801 (1990)
18. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. *Mathematical Structures in Computer Sciences* 21(4), 795–825 (2011)

# Author Index

- Accattoli, Beniamino 173  
Asperti, Andrea 240
- Barthe, Gilles 7  
Boldo, Sylvie 289  
Brassil, Matthew 126  
Bulwahn, Lukas 92
- Campbell, Brian 60  
Chan, Hing-Lun 188  
Chaudhuri, Kaustuv 208  
Coquand, Thierry 273
- Delahaye, David 76  
Doczkal, Christian 224  
Dubois, Catherine 76
- Gammie, Peter 126  
Grégoire, Benjamin 7
- Hölzl, Johannes 109
- Klein, Gerwin 126  
Kobayashi, Naoki 9  
Kunz, César 7
- Lakhnech, Yassine 7  
Lelay, Catherine 289  
Leroy, Xavier 4, 11
- Matichuk, Daniel 126  
Melquiond, Guillaume 289  
Morrisett, Greg 1  
Mörtberg, Anders 273  
Mulligan, Dominic P. 43  
Murray, Toby 126
- Nakano, Keisuke 160  
Neron, Pierre 256  
Nipkow, Tobias 109  
Norrish, Michael 188
- Popescu, Andrei 109
- Ricciotti, Wilmer 240  
Robert, Valentin 11
- Sacerdoti Coen, Claudio 43  
Shao, Zhong 143  
Siles, Vincent 273  
Smolka, Gert 224
- Tollitte, Pierre-Nicolas 76
- Vaynberg, Alexander 143
- Zanella Béguelin, Santiago 7  
Zdancewic, Steve 27  
Zhao, Jianzhou 27