Matt Kaufmann
Lawrence C. Paulson (Eds.)

LNCS 6172

Interactive Theorem Proving

First International Conference, ITP 2010
Edinburgh, UK, July 2010
Proceedings

Springer

Lecture Notes in Computer Science 6172

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

Matt Kaufmann Lawrence C. Paulson (Eds.)

Interactive Theorem Proving

First International Conference, ITP 2010
Edinburgh, UK, July 11-14, 2010
Proceedings

Springer

Volume Editors

Matt Kaufmann
University of Texas at Austin, Department of Computer Science
Austin, TX 78712, USA
E-mail: kaufmann@cs.utexas.edu

Lawrence C. Paulson
University of Cambridge, Computer Laboratory
Cambridge, CB3 0FD, UK
E-mail: lp15@cam.ac.uk

Library of Congress Control Number: 2010929576

CR Subject Classification (1998): F.3, F.4.1, D.2.4, D.2, I.2.3, D.3

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

ISSN 0302-9743
ISBN-10 3-642-14051-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14051-8 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the papers presented at ITP 2010: the First International Conference on Interactive Theorem Proving. It was held during July 11–14, 2010 in Edinburgh, Scotland as part of the Federated Logic Conference (FLoC, July 9–21, 2010) alongside the other FLoC conferences and workshops.

ITP combines the communities of two venerable meetings: the TPHOLs conference and the ACL2 workshop. The former conference originated in 1988 as a workshop for users of the HOL proof assistant. The first two meetings were at the University of Cambridge, but afterwards they were held in a variety of venues. By 1992, the workshop acquired the name *Higher-Order Logic Theorem Proving and Its Applications*. In 1996, it was christened anew as *Theorem Proving in Higher-Order Logics*, TPHOLs for short, and was henceforth organized as a conference. Each of these transitions broadened the meeting's scope from the original HOL system to include other proof assistants based on forms of higher-order logic, including Coq, Isabelle and PVS. TPHOLs has regularly published research done using ACL2 (the modern version of the well-known Boyer-Moore theorem prover), even though ACL2 implements a unique computational form of first-order logic. The ACL2 community has run its own series of workshops since 1999. By merging TPHOLs with the ACL2 workshop, we include a broader community of researchers who work with interactive proof tools.

With our enlarged community, it was not surprising that ITP attracted a record-breaking 74 submissions, each of which was reviewed by at least three Programme Committee members. The Programme Committee accepted 33 papers and asked two leading researchers, Gerwin Klein and Benjamin Pierce, to present invited lectures about their work. We were able to assemble a strong programme covering topics such as counter-example generation, hybrid system verification, translations from one formalism to another, and cooperation between tools. Several verification case studies were presented, with applications to computational geometry, unification, real analysis, etc. The tool used most in the presented papers was Coq, followed by Isabelle/HOL.

Of the 33 accepted papers, five were "proof pearls" (concise and elegant worked examples) and three were "rough diamonds" (promising ideas in an early form). All 33 papers are included in these proceedings; unlike with TPHOLs, there are no separate proceedings consisting of "Track B" papers.

We would like to thank Moshe Vardi (FLoC General Chair), Leonid Libkin and Gordon Plotkin (FLoC Co-chairs), and the other members of the FLoC Organizing Committee. David Aspinall took care of ITP local arrangements, while Michael Norrish looked after ITP's satellite workshops. We gratefully knowledge the generous support of FLoC's sponsors: the EPSRC (the UK's Engineering and Physical Sciences Research Council), NSF (the US National Science Foundation), the Association for Symbolic Logic, CADE Inc. (Conference on Automated

Deduction), Hewlett-Packard Corporation, Microsoft Research, Google Inc., and Intel Corporation.

Every aspect of the editorial process, including the production of these proceedings, was facilitated by the EasyChair conference management system. We are grateful to the EasyChair team, Andrei Voronkov and Bartek Klin, for their advice and for fixing problems quickly as they arose. We are grateful to Springer for publishing these proceedings, as they have done for TPHOLs and its predecessors since 1993.

Next year's conference, ITP 2011, will be held at the Radboud University Nijmegen, The Netherlands. This site was chosen by a ballot of the interactive theorem proving research community.

This volume is dedicated to Susan Paulson, the second editor's wife, who went into hospital around the time that these proceedings were being assembled. She had been struggling with cancer for several years. As of this writing, she was not expected to live long enough to see this volume in print.

April 2010

Matt Kaufmann
Lawrence Paulson

Conference Organization

Programme Chairs

Matt Kaufmann	University of Texas at Austin, USA
Lawrence Paulson	University of Cambridge, UK

Workshop Chair

Michael Norrish	NICTA, Australia

Programme Committee

Thorsten Altenkirch	Joe Hurd	David Pichardie
David Aspinall	Gerwin Klein	Brigitte Pientka
Jeremy Avigad	Xavier Leroy	Lee Pike
Gilles Barthe	Assia Mahboubi	Sandip Ray
Jens Brandt	Panagiotis Manolios	José Luis Ruiz Reina
Thierry Coquand	John Matthews	David Russinoff
Ruben Gamboa	J Moore	Peter Sewell
Georges Gonthier	César Muñoz	Konrad Slind
David Greve	Tobias Nipkow	Sofiène Tahar
Elsa L. Gunter	Michael Norrish	Christian Urban
John Harrison		

Local Arrangements

David Aspinall	University of Edinburgh, UK

External Reviewers

Andreas Abel
Sanaz Afshad
Mauricio Ayala-Rincón
Peter Baumgartner
Stefan Berghofer
Yves Bertot
Sandrine Blazy
David Cachera
Harsh Raju Chamarthi
James Chapman
Arthur Charguéraud
Kaustuv Chaudhuri
Cyril Cohen
Juan Manuel Crespo
Eugene Creswick
Nils Anders Danielsson
Iavor Diatchki
Lucas Dixon
Levent Erkök
Germain Faure
Andrew Gacek
Amjad Gawanmeh
Thomas Genet
Eugene Goldberg
Alwyn Goodloe
David Greenaway
Florian Haftmann
Osman Hasan
Daniel Hedin
Joe Hendrix
Brian Huffman
Ian Johnson
Andrew Kennedy
Rafal Kolanski
Alexander Krauss
John Launchbury
Hanbing Liu
Liya Liu
Luc Maranget
Andrew McCreight
James McKinna
Stephan Merz
Tarek Mhamdi
Aleksandar Nanevski
Anthony Narkawicz
Henrik Nilsson
Steven Obua
Nicolas Oury
Scott Owens
Ioana Pasca
Randy Pollack
François Pottier
Vlad Rusu
Susmit Sarkar
Carsten Schuermann
Peter-Michael Seidel
Matthieu Sozeau
Sam Staton
Rob Sumners
Andrew Tolmach
Tiphaine Turpin
Makarius Wenzel
Iain Whiteside

Table of Contents

Invited Talks

Proof Pearls

Regular Papers

Rough Diamonds

A Formally Verified OS Kernel. Now What?

Gerwin Klein

NICTA*, Australia
School of Computer Science and Engineering, UNSW, Sydney, Australia
gerwin.klein@nicta.com.au

Abstract. Last year, the L4.verified project produced a formal, machine-checked Isabelle/HOL proof that the C code of the seL4 OS microkernel correctly implements its abstract implementation. In my presentation I will summarise the proof together with its main implications and assumptions, I will describe in which kinds of systems this formally verified kernel can be used for gaining assurance on overall system security, and I will explore further future research directions that open up with a formally verified OS kernel.

1 A Formally Verified OS Kernel

Last year, we reported on the full formal verification of the seL4 microkernel from a high-level model down to very low-level C code [5].

To build a truly trustworthy system, one needs to start at the operating system (OS) and the most critical part of the OS is its *kernel*. The kernel is defined as the software that executes in the privileged mode of the hardware, meaning that there can be no protection from faults occurring in the kernel, and every single bug can potentially cause arbitrary damage. The kernel is a mandatory part of a system's *trusted computing base* (TCB)—the part of the system that can bypass security [9]. Minimising this TCB is the core concept behind *microkernels*, an idea that goes back 40 years.

A microkernel, as opposed to the more traditional *monolithic* design of contemporary mainstream OS kernels, is reduced to just the bare minimum of code wrapping hardware mechanisms and needing to run in privileged mode. All OS services are then implemented as normal programs, running entirely in (unprivileged) user mode, and therefore can potentially be excluded from the TCB. Previous implementations of microkernels resulted in communication overheads that made them unattractive compared to monolithic kernels. Modern design and implementation techniques have managed to reduced this overhead to very competitive limits.

A microkernel makes the trustworthiness problem more tractable. A well-designed high-performance microkernel, such as the various representatives of the L4 microkernel family, consists of the order of 10,000 lines of code (10 kloc).

* 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.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 1–7, 2010.

We have demonstrated that with modern techniques and careful design, an OS microkernel is entirely within the realm of full formal verification.

The approach we used was interactive, machine-assisted and machine-checked proof. Specifically, we used the theorem prover Isabelle/HOL [8]. Formally, our correctness statement is classic refinement: all possible behaviours of the C implementation are already contained in the behaviours of the abstract specification. The C code of the seL4 kernel is directly and automatically translated into Isabelle/HOL. The correctness theorem connects our abstract Isabelle/HOL specification of kernel behaviour with the C code. The main assumptions of the proof are correctness of the C compiler and linker, assembly code, hardware, and boot code.

In my presentation I will give an overview of this proof and reflect on some of the lessons learned in this project. For instance, one question with any verification is: what does the specification say, what level of detail does it contain, and why?

The current abstract specification of the seL4 kernel was written with the following goals:

- abstract from data structures and implementation, aim to specify what, not how;
- have enough detail for functional correctness proofs of applications on top of seL4;
- have enough detail for users to understand results, error conditions and error codes

Especially the latter two goals mean that the abstraction level of this specification is at points lower than one might wish for a specification. For instance, to be able to show functional correctness of code on top of seL4, we do not want to use nondeterminism too liberally in the specification. If the kernel is allowed to do either A or B nondeterministically for a certain system call, then user programs will always need to handle both cases, even though in the implementation it will usually be possible (and desired) to predict which case is taken. We have therefore usually tried to include enough data in the abstract specification layer to precisely predict kernel behaviour. This draws a small number of rather arcane implementation details to a high specification level. In hindsight it is precisely these cases that users of the kernel now find hard to understand and that are candidates for future evolvement and simplification of the seL4 API. Some of these were eliminated in early API iterations (precisely to achieve an nicer specification), but others were deemed too hard or necessary at the time.

An area where nondeterminism is more reasonable is the pure indication of failure due to implementation restrictions not visible at an abstract level. In seL4 this would mean the system call either fully succeeds or has no effect and returns with an error code. This is significantly simpler to handle for the user than two different behaviours with state change. On the other hand, this means for user programs that such system calls will always need error handling code. Hence, there is still a trade-off to be made between predictive precision and elegant abstraction.

Another way to achieve more abstraction is to focus on specific aspects of the system. We have previously published different versions of a high-level access control model of seL4 [3,2]. These models abstract from message passing, memory content, and nearly all other behaviour of the system. They only record which objects have access to which capabilities. This in turn determines which system calls a thread may perform and which objects it may access. If system calls and capabilities are additionally tagged with information flow attributes, it becomes possible to reason about the authorised information flow behaviour of a system without knowing its full functional behaviour.

Another potential abstraction of the kernel may be purely its message passing behaviour, possibly ignoring or disabling shared memory. This could be used to implement abstract communication formalisms such as the pi calculus.

2 What Now?

This section gives an overview of our current work on top of the seL4 kernel.

The main, long-term aim of this work is to produce code-level formal security proofs for large systems on the order of 1 million lines of code or more. The key idea of microkernel-based designs is to drastically reduce the trusted computing base of a system. If the system architecture explicitly takes security boundaries into account, as for example in MILS (Multiple Independent Levels of Security or Safety) systems [1], and these security boundaries are adequately enforced by the underlying OS kernel, it should be possible to reduce the amount of code that needs to be formally verified to a manageable size of a few thousand or maybe only a few hundred lines of code.

For an initial case study, we have confirmed that this is possible and feasible. The case study is a small network switching device that gives a front-end terminal access to either of two back-end networks. The job of the switch is to make sure that no data travels between the back-end networks through the switch. It is Ok for data to travel through the terminal. In this case study the user and the front-end terminal are trusted to do the right thing.

A simple Linux-based switch with web interface and user authentication will at least include the Linux kernel, a web server, and a full network stack. This already brings us into the area of millions of lines of code.

With the seL4 kernel as the bottom layer and the right architecture, we managed to isolate the security critical part into one small component with less than 2,000 lines of code. The component starts and stops Linux instances that do the actual authentication and network routing/switching work. Security is achieved by using seL4's fine-grained access control mechanism to restrict the Linux instances sufficiently such that they do not need to be trusted for the information flow property above to be true. 2,000 lines of code are small enough to be formally verified, and we have already formalised and verified the access control aspect of this trusted component as well as of the entire system. This proof shows that the modelled high-level behaviour of the trusted component achieves the security property and it shows that this component is indeed the

only trusted component of the system — all other components are assumed to try anything in their power to subvert the system.

We conducted the above proof in Isabelle/HOL. It took about 1 month of work for one person relatively new to Isabelle. We also tried the same proof in the SPIN model checker. The same person, reasonably experienced in SPIN, took about half a week to achieve the same result. This is encouraging, because it means that at least the high-level problem could be solved automatically and even the interactive proof was not very time consuming. On the other hand, even for this small system, we found it necessary to simplify and change the model to avoid state explosion in the model checker. This does not necessarily mean that SPIN would not be able to solve larger systems (our security model may just not be very suitable for model checking), but it does indicate that the process of connecting this high-level model to code will not be straight-forward.

The initial Isabelle model tried to be a faithful abstraction of the access control content of the actual system with a straightforward path to code refinement for the trusted component. The SPIN model needed custom abstractions and simplifications that (we think) were justified, but that do not have a straightforward refinement path to code any more. An interesting direction for future work will be to see if such simplifications can be automated and justified automatically by Isabelle/HOL proof.

To connect this high-level model and proof to code, the following challenges are still open: we need to show that the seL4 kernel actually implements the access control model we used in the proof and we need to show that the C code of the trusted component implements the access control behaviour that we modelled.

3 What Else?

The previous section gave a broad outline of our current research agenda. Of course, there are further useful and interesting things to be done with a formally verified OS kernel. In this section, I explore two of them.

Verified Compiler

The arguably strongest assumption of our functional correctness proof is trusting the compiler to translate seL4 correctly. We are using standard gcc for the ARM platform with optimisations. While it is almost certain that there are bugs in this version of gcc, there is sufficient hope that this particular program (seL4) does not expose them, because the seL4 implementation is sticking to well-tested parts of the C language.

Nonetheless, it would be appealing to use a verified compiler on the source code of seL4, even if it is not as strongly optimising as gcc. The obvious candidate is the CompCert project by Leroy et al [6]: an optimising C compiler, formally verified in Coq, with ARM as one possible back-end. We have not yet spent serious effort trying to use this compiler on the seL4 source, but initial investigations

show that there will be a number of challenges. The main ones are: different theorem provers, different memory models, and different C language subsets. Firstly, our proof is in Isabelle/HOL, the CompCert proof is in Coq. While it should in principle be possible to transform the proof in one system into the other, in practice both proofs are very large and an automatic transformation will quickly run into technical difficulties. Secondly, while the memory models for C used in either project are morally very close, they are technically different and it is not immediately clear how the correctness statement in one translates into the other. And finally, the C language subset used in seL4 contains features that are not strictly supported by the C standard, but are nonetheless used for kernel programming. Not all of these are covered by the CompCert compiler which is targeted more at user programs. All of these can be overcome in principle, but they will require significant effort. Even if the formalisms were not adjusted to fully fit together and the connection would therefore remain informal, one would arguably still achieve a higher level of trustworthiness by running the seL4 source through a verified compiler.

A different, similarly promising approach is the one of Slind et al [7], where the compilation process happens directly in the theorem prover. For our case the approach would probably have to be fully re-implemented in Isabelle/HOL and targeted to the SIMPL intermediate language [10] that is also the target of our parser from C into Isabelle. Performing the compilation and its correctness proof fully inside the theorem prover would have the advantage that it neatly removes the translation step from C to Isabelle from the trusted computing base. The concrete C program would merely remain a convenient syntactic representation. The actual translation would take the same formal artefact in Isabelle/HOL as source that our kernel correctness proof has as a basis. Depending on how far one would drive this approach, one could add custom compiler optimisations to particular parts of the kernel and the final product could eventually be the stream of bytes that makes up the boot image that is transferred to hardware. This would reduce the trusted part to an accurate assembly level model of the target platform — something which has already been achieved in the HOL4 prover [4].

High-Level Language Runtimes

While C is still the most widely used language in embedded and OS programming, formal C verification is far for pleasant. There are various higher-level, type safe programming languages that offer themselves more readily to formal verification. The price is not necessarily just performance, but usually also a language runtime system (memory allocation, garbage collection, etc) that is larger and arguably more complex than the whole microkernel.

This should not stop us from implementing these language runtimes on top of seL4. For one, just because they are bigger and more complex, they are not necessarily harder to verify themselves. And, even if we do not verify the language runtime, we still get the benefit of one well-tested and often-reused component that could be isolated from other parts of the system using seL4 mechanisms

and that will still provide us with the productivity and reasoning benefits of a high-level language.

Two language runtimes specifically come to mind: Haskell and JVM. Haskell (or ML), because we have made good experiences in the past with verifying large programs (e.g. the seL4 kernel) that make use of a significant subset of Haskell language features and JVM, because the JVM is arguably the most well-studied and well-formalised mainstream programming languages currently in use. The JVM has existing implementations that are certified to the highest level of Common Criteria. This means it is clear in principle how a fully verified JVM implementation and runtime can be produced.

We have conducted initial successful experiments with a Haskell runtime on top of seL4. It executes normal compiled Haskell programs that do not make use of higher-level OS services (which seL4 does not provide). It also provides a direct Haskell interface to all relevant seL4 system calls. Producing a formally verified Haskell runtime still poses a significant research challenge, but seL4 offers itself as the perfect basis for it. Given the abstract seL4 specification, it is one that could be taken on by a third party.

Acknowledgements. The security proof mentioned above was conducted almost entirely by David Greenaway with minor contributions from June Andronick, Xin Gao, and myself. The following people have contributed to the verification and/or design and implementation of seL4 (in alphabetical order): June Andronick, Timothy Bourke, Andrew Boyton, David Cock, Jeremy Dawson, Philip Derrin Dhammika Elkaduwe, Kevin Elphinstone, Kai Engelhardt, Gernot Heiser, Gerwin Klein, Rafal Kolanski, Jia Meng, Catherine Menon, Michael Norrish, Thomas Sewell, David Tsai, Harvey Tuch, and Simon Winwood.

References

1. Boettcher, C., DeLong, R., Rushby, J., Sifre, W.: The MILS component integration approach to secure information sharing. In: 27th IEEE/AIAA Digital Avionics Systems Conference (DASC), St. Paul MN (October 2008)
2. Boyton, A.: A verified shared capability model. In: Klein, G., Huuck, R., Schlich, B. (eds.) 4th WS Syst. Softw. Verification SSV'09, Aachen, Germany, October 2009. ENTCS, vol. 254, pp. 25–44. Elsevier, Amsterdam (2009)
3. Elkaduwe, D., Klein, G., Elphinstone, K.: Verified protection model of the seL4 microkernel. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 99–114. Springer, Heidelberg (2008)
4. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the armv7 instruction set architecture. In: Kaufmann, M., Paulson, L. (eds.) ITP'10. LNCS, vol. 6172, pp. 244–259. Springer, Heidelberg (2010)
5. 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 SOSP, Big Sky, MT, USA. October 2009, pp. 207–220. ACM, New York (2009)
6. Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) 33rd POPL, pp. 42–54. ACM, New York (2006)

7. Myreen, M.O., Slind, K., Gordon, M.J.C.: Extensible proof-producing compilation. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 2–16. Springer, Heidelberg (2009)
8. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
9. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proc. IEEE 63, 1278–1308 (1975)
10. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)

Proof Assistants as Teaching Assistants: A View from the Trenches

Benjamin C. Pierce

University of Pennsylvania

Abstract. Ambitious experiments using proof assistants for programming language research and teaching are all the rage. In this talk, I'll report on one now underway at the University of Pennsylvania and several other places: a one-semester graduate course in the theory of programming languages presented entirely—every lecture, every homework assignment—in Coq. This course is now in its third iteration, the course materials are becoming fairly mature, and we've got quite a bit of experience with what works and what doesn't. I'll try to give a sense of what the course is like for both instructors and students, describe some of the most interesting challenges, and explain why I now believe such machine-assisted courses are the way of the future.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, p. 8, 2010.

A Certified Denotational Abstract Interpreter*

David Cachera[1] and David Pichardie[2]

[1] IRISA / ENS Cachan (Bretagne), France
[2] INRIA Rennes – Bretagne Atlantique, France

Abstract. Abstract Interpretation proposes advanced techniques for static analysis of programs that raise specific challenges for machine-checked soundness proofs. Most classical dataflow analysis techniques iterate operators on lattices without infinite ascending chains. In contrast, abstract interpreters are looking for fixpoints in infinite lattices where widening and narrowing are used for accelerating the convergence. Smart iteration strategies are crucial when using such accelerating operators because they directly impact the precision of the analysis diagnostic. In this paper, we show how we manage to program and prove correct in Coq an abstract interpreter that uses iteration strategies based on program syntax. A key component of the formalization is the introduction of an intermediate semantics based on a generic least-fixpoint operator on complete lattices and allows us to decompose the soundness proof in an elegant manner.

1 Introduction

Static program analysis is a fully automatic technique for proving properties about the behaviour of a program without actually executing it. Static analysis is becoming an important part of modern software design, as it allows to screen code for potential bugs, security vulnerabilities or unwanted behaviours. A significant example is the state-of-the-art Astrée static analyser for C [7] which has proven some critical safety properties for the primary flight control software of the Airbus A340 fly-by-wire system. Taking note of such a success, the next question is: should we completely trust the analyser? In spite of the nice mathematical theory of program analysis and the solid algorithmic techniques available one problematic issue persists, *viz.*, the *gap* between the analysis that is proved correct on paper and the analyser that actually runs on the machine. To eliminate this gap, it is possible to merge both the analyser implementation and the soundness proof into the same logic of a proof assistant. This gives raise to the notion of *certified static analysis*, *i.e.* an analysis whose implementation has been formally proved correct using a proof assistant.

There are three main kinds of potential failures in a static analyser. The first one is (un)soundness, *i.e.* when the analyser guarantees that a program is safe but it is not. The second one deals with termination, when the analyser loops for

* Work partially supported by ANR-U3CAT grant and FNRAE ASCERT grant.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 9–24, 2010.

some entries. The third one is the problem of precision: given a choice of value approximation, the analyser may not make optimal choices when approximating some operations. For instance, a sign analyser that approximates the multiplication of two strictly positive values by the property "to be positive" is sound be not optimal since the best sound property is "to be strictly positive".[1] Only the first kind of failure is really critical. However, revealing the other bugs too late during a validation campaign may compromise the availability of the safety critical system that has to be validated in due time.

In this paper we focus on the two first classes of potential failures, that is we prove the semantic soundness of an abstract interpreter and addresses the termination problem for a challenging fixpoint iteration scheme. Most classical dataflow analysis techniques look for the least solution of dataflow (in)equation systems by computing the successive iterates of a function from a bottom element, in a lattice without infinite ascending chains. When each equation is defined by means of monotone operators, such a computation always terminates on an optimal solution, *i.e.* the least element of the lattice that satisfies all (in)equations. In contrast, abstract intepreters are generally looking for fixpoints in infinite lattices, where widening and narrowing operators are used for ensuring and accelerating the convergence. Smart iteration strategies are crucial when using such accelerating operators because they directly impact the precision of the analysis diagnostic.

This article shows how we manage to program and prove correct in Coq an abstract interpreter that uses iteration strategies based on program syntax. The purpose of the current paper is not to define widening/narrowing operators but rather use them with the right iteration strategy. We focus on a small imperative language and consider abstact interpreters that automatically infer sound invariants at each program point of a program.

Our main contribution is an elegant embedding of the Abstract Interpretation (AI) proof methodology: as far as we know, this is the first time the slogan "my abstract interpreter is correct by construction" is turned into a precise machine-checked proof. A key component of the formalization is the introduction of an intermediate semantics with respect to which the soundness of the analyser is easily proved (hence the term *by construction*). The most difficult part arises when formally linking this semantics with a standard one.

The paper is organized as follows. Section 2 introduces the problem of iteration strategies for fixpoint approximation by widening/narrowing. Section 3 briefly presents the syntax and the standard semantics of our While language. Section 4 describes the lattice theory components that are necessary for the definition of the intermediate collecting semantics described in Section 5. Section 6 presents the abstract interpreter together with its modular architecture. Then, we conclude after a discussion of related work. Except in Section 2, all formal definitions in this paper are given in Coq syntax. We heavily rely on the new type classes features [17] in order to use concise overloaded notations that

[1] This kind of "bug" is sometimes a feature in order to find a pragmatic balance between precision and algorithmic complexity.

should allow the reader to understand the formal definitions without much Coq knowledge.

2 Static Analysis with Convergence Acceleration by Widening/Narrowing

In this section, we illustrate with a simple program example the static analysis techniques that we consider. The target language is a minimalistic imperative language whose syntax is given in Section 3. The program given on Figure 1a is the core of a two-dimensional array scan with two nested loops of indexes `i` and `j`. To each program point pc in $\{1..5\}$ a pair of two intervals I_{pc} and J_{pc} is associated, each one corresponding to an over-approximation of the possible values of variables `i` and `j` at point pc.

```
    i = 0; j = 0;
1: while j < 10 {
     i = 0;
2:   while i < 10 {
       i = i + 1;
3:   };
     j = j + 1;
4: };
5:
```

(a) Program code

$$
\begin{aligned}
(I_1,\ J_1) &= ([0;0],\ [0;0]) \sqcup (I_4,\ J_4)\\
(I_2,\ J_2) &= ([0;0],\ J_1 \cap [-\infty;9]) \sqcup (I_3,\ J_3)\\
(I_3,\ J_3) &= (\mathrm{incr}^\sharp(I_2),\ J_2)\\
(I_4,\ J_4) &= (I_2 \cap [10;+\infty],\ \mathrm{incr}^\sharp(J_2))\\
(I_5,\ J_5) &= (I_1,\ J_1 \cap [10;+\infty])
\end{aligned}
$$

(b) Analysis equations

Fig. 1. An example of interval analysis

The analysis is specified by mutually recursive equations relating the intervals to each other. Figure 1b lists the equations corresponding to our example. The domain of intervals is equipped with a special bottom element $\bot$, and abstract operators like intersection ($\cap$), convex union ($\sqcup$) or abstract incrementation ($\mathrm{incr}^\sharp$ defined by $\mathrm{incr}^\sharp([a,b]) = [a+1, b+1]$ and extended naturally to infinite values). These equations can be summarized as $\boldsymbol{X} = \boldsymbol{F}(\boldsymbol{X})$, where $\boldsymbol{X}$ denotes the vector $(X_1, X_2, X_3, X_4, X_5)$ and each X_i denotes a pair of intervals. The components of $\boldsymbol{F}$ will be denoted by $F_i, i \in \{1..5\}$ in the rest of this section.

The set of intervals on integers forms a lattice ordered by inclusion, where the lub and glb operators are the convex union and intersection, respectively. Ideally, the result of the analysis should be the least fixpoint of the set of equations. However, the lattice of intervals is of infinite height, which generally prevents us from computing a solution in finite time with a simple Kleene iteration. For instance, $\emptyset \subset [0,0] \subset [0,1] \subset [0,2] \subset \cdots \subset [0,n] \subset \ldots$ is an infinite increasing chain. Instead of computing a least fixpoint, we can accommodate ourselves with an over-approximation, keeping correction in mind, but losing optimality.

The solution proposed by P. and R. Cousot [6] consists in accelerating the ascending iteration, thus reaching a post-fixpoint, but not necessarily the least one. This is done by using a binary *widening* operator $\triangledown$, that extrapolates both of its arguments, and use an iteration of the following form: $x_0 = \bot, x_{n+1} = x_n \triangledown f(x_n)$. Intuitively, at the n-th iteration, the n-th iterate of the function is compared to the preceeding value, in order to conjecture some possible over-approximation of the limit of the iteration sequence. In the infinite chain above, we would like to replace interval $[0, n]$ by $[0, +\infty]$ after a finite (preferably small) number of iterations. We would like for instance that $[0, 1] \triangledown [0, 2] = +\infty$.

When a post-fixpoint is reached, a new iteration starting from this point may give a better solution, closer to the least fixpoint. The same termination issues appear, and may be treated by the use of a narrowing operator $\vartriangle$.

The equations of the analysis are thus augmented with a set W of program points where widenings or narrowings are performed. We now have to determine a strategy for computing a fixpoint, *i.e.* an oracle for choosing an equation during the following standard *chaotic iteration*.

- Start with $\boldsymbol{X} = (\bot, \bot, \bot, \bot, \bot)$.
- Repeat until a post-fixpoint is reached the following steps
  - Choose (oracle) a point pc,
  - if pc is a widening/narrowing point, replace X_{pc} by $X_{pc} \triangledown F_{pc}(\boldsymbol{X})$ otherwise by $X_{pc} \sqcup F_{pc}(\boldsymbol{X})$.
- Repeat until stabilization the following steps
  - Choose (oracle) a point pc,
  - if pc is a widening/narrowing point, replace X_{pc} by $X_{pc} \vartriangle F_{pc}(\boldsymbol{X})$ otherwise by $X_{pc} \sqcap F_{pc}(\boldsymbol{X})$.

Several technical difficulties appear with iteration strategies. First, widening/-narrowing points must be placed in order to cover all equations dependency cycles, otherwise the iteration may not terminate. On the contrary, using too many points may decrease the precision of the final result. Secondly, the iteration strategy (oracle) has a direct impact on the precision of the result because of the non-monotone nature of widening/narrowing operators.

To illustrate these points, let us consider two strategies for our analysis example. Both strategies have widening/narrowing points at 1 and 2, the loop entries. The first one examines all equations in sequence with order 1 2 3 4 5 until stabilization. The second one has an inner iteration: it waits until the inner loop $(2\,3)^*$ stabilizes before computing equations 4 and 5. Results are displayed in Figure 2.

We give for each strategy the details of the ascending iteration with widening for points 1 and 2, and the final result after widening and narrowing iterations for all control points. This example shows the importance of choosing a good strategy: the second strategy which includes an "inner" iteration between points 2 and 3 yields a more precise result. This comes from the widenings performed at point 2: with strategy $(1\,2\,3\,4\,5)^*$ (Figure 2a), a widening is performed at the second iteration round on both variables `i` and `j`; as J_2 depends on J_1 computed in the same round, J_2 is "widened" to $[0, +\infty]$, even if the inner loop only affects

Ascending iteration for points 1 and 2			
iteration round	1	2	3
I_1	$[0,0]$	$[0,0]$	$[0,+\infty]$
J_1	$[0,0]$	$[\mathbf{0},+\infty]$	$[0,+\infty]$
I_2	$[0,0]$	$[0,+\infty]$	$[0,+\infty]$
J_2	$[0,0]$	$[0,+\infty]$	$[0,+\infty]$

Post-fixpoint after ascending and descending iterations				
$I_1 = [0,10]$	$I_2 = [0,10]$	$I_3 = [1,10]$	$I_4 = [10,10]$	$I_5 = [0,10]$
$J_1 = [0,+\infty]$	$J_2 = [0,+\infty]$	$J_3 = [0,+\infty]$	$J_4 = [1,+\infty]$	$J_5 = [10,+\infty]$

(a) Strategy $(1\,2\,3\,4\,5)^*$

Ascending iteration for points 1 and 2					
iteration round	1	2	3	4	5
I_1	$[0,0]$	$[0,0]$	$[0,0]$	$[0,+\infty]$	$[0,+\infty]$
J_1	$[0,0]$	$[0,0]$	$[0,0]$	$[0,+\infty]$	$[0,+\infty]$
I_2	$[0,0]$	$[0,+\infty]$	$[0,+\infty]$	$[0,+\infty]$	$[0,+\infty]$
J_2	$[0,0]$	$[0,0]$	$[0,0]$	$[0,9]$	$[0,9]$

Post-fixpoint after ascending and descending iterations				
$I_1 = [0,10]$	$I_2 = [0,10]$	$I_3 = [1,10]$	$I_4 = [10,10]$	$I_5 = [0,10]$
$J_1 = [\mathbf{0},\mathbf{10}]$	$J_2 = [\mathbf{0},\mathbf{9}]$	$J_3 = [\mathbf{0},\mathbf{9}]$	$J_4 = [\mathbf{1},\mathbf{10}]$	$J_5 = [\mathbf{10},\mathbf{10}]$

(b) Strategy $(1\,(2\,3)^*\,4\,5)^*$

Fig. 2. Example of iteration strategies

`i`; on the contrary, with strategy $(1\,(2\,3)^*\,4\,5)^*$ (Figure 2b), the value of J_1 is not modified along the inner iteration $(2\,3)^*$, so that J_2 is not "widened" too early.

Strategy $(1\,(2\,3)^*\,4\,5)^*$ is not taken at random: it exactly follows the syntactic structure of the program, computing fixpoints as would a denotational semantics do. This is why we consider this kind of strategy in this paper.

3 Language Syntax and Operational Semantics

The analyser we formalize here is taken from an analysis previously designed by Cousot [5]. We consider a minimal WHILE language whose concrete COQ syntax is given below. Programs are labelled[2] with elements of type `word`, which plays a special role in all our development. It is the type of COQ binary numbers with at most 32 bits and is hence inhabited with a finite number of objects. This property is crucial in order to ensure termination of fixpoint iteration on arrays indexed by keys of type `word` (*cf.* Section 6). A program is made from a main instruction, an end label and a set of local variables. The syntax of instructions contains assignments of a variable by a numeric expression, conditionals, while loops.

[2] Contrary to the example given in Section 2, we give a label to each instruction.

```
Definition var := word.
Definition pp := word.
Inductive op := Add | Sub | Mult.
Inductive expr :=
 Const (n:Z) | Unknown | Var (x:var) | Numop (o:op)(e1 e2:expr).
Inductive comp := Eq | Lt.
Inductive test :=
| Numcomp (c:comp) (e1 e2:expr) | Not (t:test)
| And (t1 t2:test) | Or (t1 t2:test).
Inductive instr :=
  Assign (p:pp) (x:var) (e:expr) | Skip (p:pp)
| Assert (p:pp) (t:test) | If (p:pp) (t:test) (b1 b2:instr)
| While (p:pp) (t:test) (b:instr) | Seq (i1:instr) (i2:instr).
Record program := { p_instr:instr;  p_end: pp; vars: list var }.
```

We give to this language a straightforward small-step operational semantics. An environment maps variable names to numerical values. A configuration is either a final environment or an intermediate configuration. The operational semantics takes the form of a judgment `(sos p k (i,ρ) s)` that reads as follows: for a program `p` there is a one-step transition between an intermediate configuration `(i,ρ)` (an instruction and an environment) towards configuration `s`. Predicate `(subst ρ1 x n ρ2)` expresses that the environment `ρ2` is the result of the substitution of `x` by `n` in `ρ1`. The element `k` records the kind of transition that is taken. It is a technical trick that will be used and explained later in Section 5. At last, we build the reflexive transitive closure `sos_star` of `sos`.

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
Inductive sos (p:program) : Kind → (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (KAssign x e) (Assign l x e,ρ1) (Final ρ2)
[...]
| sos_while_true : ∀ l t b ρ,
  sem_test p ρ t true →
  sos p (KAssert t) (While l t b,ρ) (Inter (Seq b (While l t b)) ρ)
| sos_while_false : ∀ l t b ρ,
  sem_test p ρ t false →
  sos p (KAssert (Not t)) (While l t b,ρ) (Final ρ)
| sos_seq1 : ∀ k i1 i2 ρ ρ',
  sos p k (i1,ρ) (Final ρ') →
  sos p (KSeq1 i (first i2)) (Seq i1 i2,ρ) (Inter i2 ρ')
| sos_seq2 : ∀ k i1 i1' i2 ρ ρ',
  sos p k (i1,ρ) (Inter i1' ρ') →
  sos p (KSeq2 k) (Seq i1 i2,ρ) (Inter (Seq i1' i2) ρ').
Inductive sos_star (p:program) : (instr*env) → config → Prop :=
| sos_star0 : ∀ i ρ, sos_star p (i,ρ) (Inter i ρ)
| sos_star1 : ∀ k s1 s2, sos p k s1 s2 → sos_star p s1 s2
| sos_trans : ∀ k s1 i ρ s3,
  sos p k s1 (Inter i ρ) → sos_star p (i,ρ) s3 → sos_star p s1 s3.
```

The soundness theorem of the analyser is expressed in terms of labelled reachable states in `(pp * env)`. A special label `p_end` is attached to final environments. Function `first` recursively computes the left-most label of an instruction.

```
Inductive reachable_sos (p:program) : pp*env → Prop :=
| reachable_sos_intermediate : ∀ ρ0 i ρ,
  sos_star p (p_instr p,ρ0) (Inter  i ρ) →
  reachable_sos p (first i,ρ)
| reachable_sos_final : ∀ ρ0 ρ,
  sos_star p (p_instr p,ρ0) (Final ρ) →
  reachable_sos p (p_end p,ρ).
```

4 Lattice Theory Intermezzo

Abstract Interpretation heavily relies on lattice theory to formalize semantic notions and approximation of properties. In order to define the suitable intermediate semantics on our WHILE language in the next section, we need to define a least fixpoint operator that is defined on complete lattices. We then introduce several packed structures for equivalence relations, partial orders and complete lattices. We use here type classes that give elegant notations for defining COQ records with implicit arguments facilities. Coercions are also introduced in order to define `(Equiv.t A)` as subclass of `(Poset.t A)` (notation `:>` in the field `eq` of the class type `Poset.t`) and view subset components of type `subset A` as simple predicates on `A` (command **`Coercion`** `charact : subset >->`**`Funclass`**). Some parameters are given with curly braces `` `{...} `` in order to declare them as implicit. All these features make the formal definition far more elegant and concise. For example, in the type declaration of the field `join_bound`, the term `subset A` hides an object `(Poset.eq A porder)` of type `(Equiv.t A)` which is automatically taken from the field `porder : Poset.t A` of the class type `CompleteLattice.t` and the coercion between `(Equiv.t A)` and `(Poset.t A)`.

We also introduce definitions for lattices (type class `Lattice.t`) with the corresponding overloaded symbols $\bot$, $\top$, $\sqcap$, $\sqcup$ but do not show them here, due to space constraints.

```
Module Equiv.
Class t (A:Type) : Type :=
 { eq : A → A → Prop;
   refl : ∀ x, eq x x; sym : [...]; trans : [...] }.
End Equiv.
Notation "x == y" := (Equiv.eq x y) (at level 40).

Module Poset.
Class t A : Type :=
 { eq :> Equiv.t A;
   order : A → A → Prop;
   refl : [...];  antisym : [...]; trans : [...] }.
End Poset.
```

```
Notation "x ⊑ y" := (Poset.order x y) (at level 40).

Class subset A `{Equiv.t A} : Type := SubSet {
 charact : A → Prop;
 subset_comp_eq : ∀ x y:A, x==y →  charact x → charact y}.
Coercion charact : subset >-> Funclass.

Module CompleteLattice.
Class t (A:Type) : Type := {
 porder :> Poset.t A;
 join : subset A → A;
 join_bound : ∀x:A, ∀f:subset A, f x → x ⊑ join f;
 join_lub : ∀f:subset A, ∀z,(∀ x:A, f x → x ⊑ z) → join f ⊑ z;}.
End CompleteLattice.
```

Several canonical structures can be defined for these types. They will be automatically introduced by the inference system when objects of these types will be wanted by the type checker.

```
Notation "'𝒫' A" := (A → Prop) (at level 10).
Instance PowerSetCL A : CompleteLattice.t (𝒫 A) := [...]
Instance PointwiseCL A L `{CompleteLattice.t L} :
                        CompleteLattice.t (A → L) := [...]
```

Monotone functions are defined with a dependent pair and a coercion.

```
Class monotone A `{Poset.t A} B `{Poset.t B} : Type := Mono {
  mon_func : A → B;
  mon_prop : ∀ a1 a2, a1 ⊑ a2 → (mon_func a1) ⊑ (mon_func a2)}.
Coercion mon_func : monotone >-> Funclass.
```

We finish this section with the classical Knaster-Tarski theorem.

```
Definition lfp `{CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).

Section KnasterTarski.
  Variable L : Type.
  Variable CL : CompleteLattice.t L.
  Variable f : monotone L L.
  Lemma lfp_fixpoint : f (lfp f) == lfp f. [...]
  Lemma lfp_least_fixpoint : ∀ x, f x == x → lfp f ⊑ x. [...]
  Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f. [...]
  Lemma lfp_least_postfixpoint : ∀x, f x ⊑ x → lfp f ⊑ x. [...]
End KnasterTarski.
```

5 An Intermediate Collecting Semantics

We now reach the most technical part of our work. One specific feature of the AI methodology is to give program semantics and program static analyses the same shape. To that purpose, a *collecting* semantics is used instead of the standard

semantics. Its purpose is still to express the dynamic behaviour of programs but it takes a form closer to that of a static analysis, namely being expressed as a least fixpoint of an equation system. In this work, we want to certify an abstract interpreter that inductively follows the syntax of programs and iterates each loop until convergence. We will thus introduce a collecting semantics that follows the same strategy, but computes on concrete properties.

In all this section we fix a program `prog:program`. We first introduce three semantic operators. Operator `assign` is the strongest post-condition of the assignment of a variable by an expression. Operator `assert` is the strongest post-condition of a test. The collecting semantics binds each program point to a property over reachable environments so that the semantic domain for this semantics is `pp →𝒫(A)`. An element in this domain can be updated with the function `Esubst`. Note that it is a weak update since we take the union with the previous value.

```
Definition assign (x:var) (e:expr) (E:𝒫(env)) : 𝒫(env) :=
 fun ρ => ∃ρ', ∃n, E ρ' ∧ sem_expr prog ρ' e n ∧ subst ρ' x n ρ.
Definition assert (t:test) (E:𝒫(env)) : 𝒫(env) :=
  fun ρ => E ρ ∧ sem_test prog ρ t true.
Definition Esubst `(f:pp → 𝒫(A)) (k:pp) (v:𝒫(A)) : pp → 𝒫(A) :=
  fun k' => if pp_eq k' k then (f k) ⊔ v else f k'.
Notation "f +[ x ↦ v ]" := (Esubst f x v) (at level 100).
```

The collecting semantics is then defined by induction on the program syntax. Function `(Collect i l)` computes for an instruction `i` and a label `l`, a monotone predicate transformer such that for each initial property `Env:𝒫(env)`, all states `(l',ρ)` that are reachable by the execution of `i` from a state in `Env`, satisfy `(Collect i l Env l' ρ)` where `l'` is either a label in `i` or is equal to `l`. The label `l` should represent the next label after `i` in the whole program. The monotony property of this predicate transformer returned by `(Collect i l)` is crucial if want to be able to use the `lfp` operator of the previous section that takes only monotone functions as argument. For each instruction, we hence build a term of the form `(Mono F _)` with `F` a function and `_` a "hole" for the proof that ensures the monotony of `F`. All these holes give rise to proof obligations that are generated by the **Program** mechanism and must be interactively discharged after the definition of `Collect`.

```
Program Fixpoint Collect (i:instr) (l:pp):
                              monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Skip p => Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ Env]) _
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | Assert p t =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assert t Env]) _
  | If p t i1 i2 =>
    Mono (fun Env =>
          let C1 := Collect i1 l (assert t Env) in
          let C2 := Collect i2 l (assert (Not t) Env) in
```

```
            (C1 ⊔ C2) +[p ↦ Env]) _
  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I))
                  +[p ↦ I] +[l ↦ assert (Not t) I]) _
  | Seq i1 i2 =>
    Mono (fun Env => let C := (Collect i1 (first i2) Env) in
                        C ⊔ (Collect i2 l (C (first i2)))) _
end.
```

The `While` instruction is the most complex case. It requires to compute a predicate `I:𝒫(env)` that is a loop invariant bound to the label `p`. It is the least fixpoint of the monotone operator `iter` defined below.

```
Program Definition iter (Env:𝒫(env))
  (F:monotone (𝒫(env)) (pp → 𝒫(env)))
  (t:test) (l:pp) : monotone (𝒫(env)) (𝒫(env)) :=
  (Mono (fun X => Env ⊔ (F (assert t X) l)) _).
```

We hence compute the least fixpoint `I` of the following equation:

```
I == Env ⊔ (Collect i p (assert t I) p)
```

The invariant is the union of the initial environment (at the entry of the loop) and the result of the transformation of `I` by two predicate transformers: `assert t` takes into account the entry in the loop, and the recursive call to `(Collect i p)` collects all the reachable environments during the execution of the loop body `i` and select those that are bound to the label `p` at the loop header.

Such a semantics is sometimes taken as *standard* in AI works but here, we precisely prove its relationship with the more standard semantics of Section 3. For that purpose, we establish the following theorem.

```
Theorem sos_star_implies_Collect : ∀ i1 ρ1 s2,
  sos_star prog (i1,ρ1) s2 →
    match s2 with
      | Inter i2 ρ2 => ∀l:pp, ∀Env:𝒫(env),
          Env ρ1 → Collect i1 l Env (first i2) ρ2
      | Final ρ2 => ∀l:pp, ∀Env:𝒫(env),
          Env ρ1 → Collect i1 l Env l ρ2
    end.
```

This theorem is proved by induction on the hypothesis `sos_star prog s1 s2`. The first case corresponds to `s2=Inter i1 ρ1` and is proved thanks using the fact that `Collect` is extensive.

```
Lemma Collect_extensive : ∀ i, ∀ Env:𝒫(env), ∀l,
  Env ⊑ Collect i l Env (first i).
```

The second case corresponds to `sos prog s1 s2`. It is proved thanks to the following lemma.

```
Lemma sos_implies_Collect : ∀ k i1 ρ1 s2,
  sos prog k (i1,ρ1) s2 →
    match s2 with
      | Inter i2 ρ2 => ∀l:pp, ∀Env:𝒫(env),
          Env ρ1 → Collect i1 l Env (first i2) ρ2
      | Final ρ2 => ∀l:pp, ∀Env:𝒫(env),
          Env ρ1 → Collect i1 l Env l ρ2
    end.
```

The last case corresponds to the existence of an intermediate state `(i,ρ)` such that `sos prog k (i1, ρ1) (Inter i ρ)` and `sos_star prog (i, ρ) s2` and the induction hypothesis holds for `(Collect i)`. In order to connect this induction hypothesis with the goal that deals with `(Collect i1)`, we prove the following lemma.

```
Lemma sos_transfer_incl : ∀ k i1 ρ1 i2 ρ2,
  sos prog k (i1,ρ1) (Inter i2 ρ2) →
  ∀Env:𝒫(env), ∀l_end,
    Collect i2 l_end (transfer k Env) ⊑ Collect i1 l_end Env.
```

Here `(transfer k)` is a suitable predicate transformer such that the following lemma holds.

```
Lemma sos_transfer : ∀ k i1 ρ1 s2,
  sos prog k (i1,ρ1) s2 →
    match s2 with
      | Inter i2 ρ2 => ∀Env:𝒫(env), Env ρ1 → (transfer k Env) ρ2
      | Final ρ2  => ∀Env:𝒫(env), Env ρ1 → (transfer k Env) ρ2
    end.
```

It is defined by the following recursive function using the information `k:Kind` that we have attached to each semantic rule in Section 3.

```
Fixpoint transfer (k:Kind) (Env:𝒫(env)) : 𝒫(env) :=
  match k with
    | KAssign x e => assign x e Env
    | KSkip => Env
    | KAssert t => assert t Env
    | KSeq1 i l => Collect i l Env l
    | KSeq2 k => transfer k Env
  end.
```

This ends the proof of theorem `sos_star_implies_Collect`, from which we can easily deduce that each reachable state in the standard semantics is reachable with respect to the collecting semantics.

```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in Collect p (p_instr p) (p_end p) (⊤) k env.
Theorem reachable_sos_implies_reachable_collect :
  ∀ p s, reachable_sos p s → reachable_collect p s.
```

6 A Certified Abstract Interpreter

We now design an abstract interpreter that, instead of executing a program on environment properties as the previous collecting semantics do, computes on an abstract domain with a lattice structure. Such a structure is modelled with a type class `AbLattice.t` that packs the same standard components as in `Lattice.t` but also decidable tests for equality and partial order, and widening/narrowing operators. It is equipped with overloaded notations $\sqsubseteq^\sharp$, $\sqcap^\sharp$, $\sqcup^\sharp$, $\bot^\sharp$. This abstract lattice structure has been presented in an earlier paper [14]. The lattice signature contains a well-foundedness proof obligation which ensures termination of a generic post-fixpoint iteration algorithm.

```
Definition approx_lfp : ∀ `{AbLattice.t t}, (t → t) → t := [...]
Lemma approx_lfp_is_postfixpoint : ∀ `{AbLattice.t t} (f:t → t),
    f (approx_lfp f) ⊑♯ (approx_lfp f).
```

A library is provided to build complex lattice objects with various functors for products, sums, lists and arrays [14]. Arrays are defined by means of binary trees whose keys are elements of type `word`. The corresponding functor `ArrayLattice` given below relies on the finiteness of `word` to prove the convergence of the pointwise widening/narrowing operators on arrays.

```
Instance ArrayLattice t `{L:AbLattice.t t}: AbLattice.t(array t).
```

We connect concrete and abstract lattices with concretization functions that enjoy a monotony property together with a meet morphism property. This formalization choice is motivated by our previous study of embedding AI framework in the constructive logic of Coq [13].

```
Module Gamma.
Class t a A `{Lattice.t a} `{AbLattice.t A} : Type := {
  γ : A → a;
  γ_monotone : ∀ N1 N2:A, N1 ⊑♯ N2 → γ N1 ⊑ γ N2;
  γ_meet_morph : ∀ N1 N2:A, γ N1 ⊓ γ N2 ⊑ γ (N1 ⊓♯ N2)
}.
End Gamma.
Coercion Gamma.γ : Gamma.t >-> Funclass.
```

Using the new Coq type class feature we have extended our previous lattice library with *concretization functors* in order to build concretization operators in a modular fashion. We show below the signature of the array functor. This kind of construction was difficult with Coq modules that are not first class citizens, whereas it is often necessary to let concretizations depend on elements as programs.

```
Instance GammaFunc `{Gamma.t a A} : Gamma.t (word → a)(array A).
```

Our abstract interpreter is parameterized with respect to an abstraction of program environments given below. The structure encloses a concretization operator mapping environment properties to an abstract lattice, two correct approximations of the predicate transformers `Collect.assign` and `Collect.assert` defined in Section 5, and an approximation of the "don't know" predicate.

```
Module AbEnv.
Class t `(L:AbLattice.t A) (p:program) : Type := {
  gamma :> Gamma.t (𝒫 env) A;
  assign : var → expr → A → A;
  assign_correct : ∀ x e,
    (Collect.assign p x e) ∘ gamma ⊑ gamma ∘ (assign x e);
  assert : test → A → A;
  assert_correct : ∀ t,
    (Collect.assert p t) ∘ gamma ⊑ gamma ∘ (assert t);
  top : A;
  top_correct : ⊤ ⊑ gamma top
}.
End AbEnv.
```

The abstract interpreter `AbSem` is then defined in a section where we fix a program and an abstraction of program environments. Its definition perfectly mimics the collecting semantics. We use the abstract counterpart `F +[p ↦Env]♯` of the operator `Esubst` that has been defined in Section 5. The main difference is found for the `While` instruction where we don't use a least fixpoint operator but the generic post-fixpoint solver `approx_lfp`.

```
Section prog.
Variable (t : Type) (L : AbLattice.t t)
         (prog : program) (Ab : AbEnv.t L prog).

Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
match i with
| Skip p => fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Env]♯
| Assign p x e =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
| Assert p t =>
  fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assert t Env]♯
| If p t i1 i2 => fun Env =>
       let C1 := AbSem i1 l (Ab.assert t Env) in
       let C2 := AbSem i2 l (Ab.assert (Not t) Env) in
           (C1 ⊔♯ C2) +[p ↦ Env]♯
| While p t i => fun Env =>
  let I := approx_lfp (fun X => Env ⊔♯
                         (get (AbSem i p (Ab.assert t X)) p)) in
 (AbSem i p (Ab.assert t I))+[p ↦ I]♯+[l ↦ Ab.assert (Not t) I]♯
| Seq i1 i2 =>  fun Env =>
  let C := (AbSem i1 (first i2) Env) in
    C ⊔♯ (AbSem i2 l (get C (first i2)))
end.
```

This abstract semantics is then proved correct with respect to the collecting semantics `Collect` and the canonical concretization operator on arrays. The proof is particularly easy because `Collect` and `AbSem` share the same shape.

```
Definition γ : Gamma.t (word → 𝒫(env)) (array t) := GammaFunc.
```

```
Lemma AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (Ab.γ Env) ⊑ γ (AbSem i l_end Env).
```

At last we define the program analyser and prove that it computes an over-approximation of the reachable states of a program. This a direct consequence of the previous lemma. Note that this final theorem deals with the standard operational semantics proposed in Section 3. The collecting semantics in only used as an intermediate step in the proof.

```
Definition analyse : array t :=
  AbSem prog.(p_instr) prog.(p_end) (Ab.top).
Theorem analyse_correct : ∀ k env,
  reachable_sos prog (k,env) → Ab.γ (get analyse k) env.
```

In order to instantiate the environment abstraction, we provide a functor that builds a non-relational abstraction from any numerical abstraction by binding a numerical abstraction to each program variable.

```
Instance EnvNotRelational `(NumAbstraction.t L) (p:program) :
  AbEnv.t (ArrayLattice L) p.
```

Due to the lack of space, the type `NumAbstraction.t` is not described in this paper. We have instantiated it with interval, sign and congruence abstractions. The different instances of the analyser can be extracted to OCaml code and run on program examples[3].

7 Related Work

The analyser we have formalized here is taken from lecture notes by Patrick Cousot [5,4]. We follow only partly his methodology here. Like him, we rely on a collecting semantics which gives a suitable semantic counterpart to the abstract interpreter. This semantics requires elements of lattice theory that, as we have demonstrated, fit well in the Coq proof assistant. One first difference is that we don't take this collecting semantics as standard but formally link it with a standard small-step semantics. A second difference concerns the proof technique. Cousot strives to manipulate Galois connections, which are the standard abstract interpretation constructs used for designing abstract semantics. Given a concrete lattice of program properties and an abstract lattice (on which the analyser effectively computes), a pair of operators (α, γ) is introduced such that $\alpha(P)$ provides the best abstraction of a concrete property P, and $\gamma(P^\sharp)$ is the least upper bound of the concrete properties that can be soundly approximated by the abstract element $P^\sharp$. With such a framework it is possible to express the most precise correct abstract operator $f^\sharp = \alpha \circ f \circ \gamma$ for a concrete f. Patrick Cousot in another set of lecture notes [5] performs a systematic derivation of abstract transfers functions from specifications of the form $\alpha \circ f \circ \gamma$. We did not follow this kind of symbolic manipulations here because they will require much proof effort: each manipulation of α requires to prove a property of optimal approximation.

[3] The Coq development is available at
`http://irisa.fr/celtique/pichardie/ext/itp10`

This is only useful for tracking the third category of static analysis failures we have mentioned in the introduction of this paper.

Our abstract interpreter may appear as a toy example but it is often presented [16] as the core of the Astrée static analyser for C [7]. The same iteration strategy is used and the restrictions on the C language that are common in many critical embedded systems (no recursion, restricted use of pointers) allow Astrée to concentrate mainly on While static analysis techniques. We are currently studying how we might formally link our current abstract interpreter with the formal semantics [3] of the CompCert project. This project is dedicated to the certification of a realistic optimizing C compiler and has so far [10] only be interested in data flow analyses without widening/narrowing techniques.

The While language has been the subject of several machine-checked semantics studies [8,12,2,9] but few have studied the formalization of abstract interpretation techniques. A more recent approach in the field of semantics formalization is the work of Benton *et al.* [1] which gives a Coq formalization of cpos and of the denotational semantics of a simple functional language. A first attempt of a certified abstract interpreter with widening/narrowing iteration techniques has been proposed by Pichardie [13]. In this previous work, the analyser was directly generating an (in)equation system that was solved with a naive round-robin strategy as in Figure 2a. The soundness proof was performed with respect to an ad-hoc small-step semantics. Bertot [2] has formalized an abstract interpreter whose iteration strategy is similar to ours. His proof methodology differs from the traditional AI approach since the analyser is proved correct with respect to a weakest-precondition calculus. The convergence technique is more ad hoc than the standard widening/narrowing approach we follow. We believe the inference capability of the abstract interpreters are similar but again, our approach follows more closely the AI methodology with generic interfaces for abstraction environments and lattice structures with widening/narrowing. We hence demonstrate that the Coq proof assistant is able to follow more closely the textbook approach.

8 Conclusion and Perspectives

We have presented a certified abstract interpreter for a While language which is able to automatically infer program invariants. We have in particular studied the syntax-directed iteration strategy that is used in the Astrée tool. A similar interpreter had been proposed earlier [2] but our approach follows more closely the AI methodology. The key ingredient of the formalization is an intermediate collecting semantics which is proved conservative with respect to a classical structural operational semantics [15].

The current work is a first step towards a global objective of putting in the Coq proof assistant most of the advanced static analysis techniques that are used in an analyser like Astrée. We could enhance it by function calls, that would be very easy to handle if we avoid recursion: as our abstract interpreter is denotational, *i.e.* is able to take any abstract property as input of a block and compute a sound over-approximation of the states reachable from it. In this work we have

computed an invariant for each program point but Astrée spares some computations keeping as few invariants as possible during iteration, and we should also consider that approach. These topics only concern the generic interpreter but we will have to combine this core interpreter with suitable abstract domains like octogons for floating-point arithmetic [11] or fine trace abstractions [16].

References

1. Benton, N., Kennedy, A., Varming, C.: Some domain theory and denotational semantics in Coq. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 115–130. Springer, Heidelberg (2009)
2. Bertot, Y.: Structural abstract interpretation, a formal study in Coq. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) ALFA. LNCS, vol. 5520, pp. 153–194. Springer, Heidelberg (2009)
3. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reasoning 43(3), 263–288 (2009)
4. Cousot, P.: Visiting Professor at the MIT Aeronautics and Astronautics Department, Course 16.399: Abstract Interpretation
5. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77, pp. 238–252. ACM Press, New York (1977)
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
8. Gordon, M.J.C.: Mechanizing programming logics in higher-order logic. In: Current Trends in Hardware Verfication and Automatic Theorem Proving, pp. 387–439. Springer, Heidelberg (1988)
9. Leroy, X.: Mechanized semantics, with applications to program proof and compiler verification, lecture given at the, Marktoberdorf summer school (2009)
10. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Proc. of POPL'06, pp. 42–54. ACM Press, New York (2006)
11. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
12. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing 10, 171–186 (1998)
13. Pichardie, D.: Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés. PhD thesis, Université Rennes 1 (2005) (in french)
14. Pichardie, D.: Building certified static analysers by modular construction of well-founded lattices. In: Proc. of FICS'08. ENTCS, vol. 212, pp. 225–239 (2008)
15. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming 60-61, 17–139 (2004)
16. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5) (2007)
17. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)

Using a First Order Logic to Verify That Some Set of Reals Has No Lesbegue Measure

John Cowles and Ruben Gamboa

University of Wyoming
Laramie, WY, USA
{cowles,ruben}@cs.uwyo.edu

Abstract. This paper presents a formal proof of Vitali's theorem that not all sets of real numbers can have a Lebesgue measure, where the notion of "measure" is given very general and reasonable constraints. A careful examination of Vitali's proof identifies a set of axioms that are sufficient to prove Vitali's theorem, including a first-order theory of the reals as a complete, ordered field, "enough" sets of reals, and the axiom of choice. The main contribution of this paper is a positive demonstration that the axioms and inference rules in ACL2(r), a variant of ACL2 with support for nonstandard analysis, are sufficient to carry out this proof.

1 Introduction

The notion of Lebesgue measure, reviewed in section 2, generalizes the concept of length for sets of real numbers. It is a surprising result in real analysis that not all sets of real numbers can be adequately assigned a measure, provided that the notion of "measure" conforms to some reasonable constraints, such as the measure defined by Lebesgue. This paper presents a formal proof of this result, which is due to Vitali.

The formal proof is carried out in ACL2(r), a variant of ACL2 that offers support for nonstandard analysis. Although ACL2(r) adds built-in support for key concepts from nonstandard analysis, such as "classical" and "standard part", it retains the strengths and limitations of ACL2. In particular, it is a strictly first-order theory, with only limited support for quantifiers. One important logical feature of ACL2 and ACL2(r) is the introduction of constrained functions, which allows these theorem provers to reason about classes of functions, e.g., all continuous functions, in a strictly first-order setting. This is similar to the mathematical practice of reasoning about a generic continuous function, in order to establish a theorem that applies to all continuous functions. Moreover, ACL2 and ACL2(r) support a definition principle that allows the introduction of "choice" functions via a Skolem axiom. This powerful definitional principle works as follows. Let ϕ be a formula whose only free variables are v, x_1, x_2, ..., x_n. The Skolem axiom introducing f from ϕ with respect to v is

$$\phi \Rightarrow \textbf{let } v = f(x_1, x_2, \ldots, x_n) \textbf{ in } \phi$$

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 25–34, 2010.

What this axiom states is that the function f can "choose" an appropriate v for a given $x_1, x_2, \ldots, x_n$, provided such a choice is at all possible. This principle was recently "strengthened" in ACL2, and the strong version of this principle takes the place of the Axiom of Choice in the formal proof of Vitali's Theorem. We do not believe that this proof could have been carried out using the original version of this definitional principle. Perhaps surprisingly, these two definitional principles are conservative in the logic of ACL2. This logic is precisely described in [7,3].

The paper is organized as follows. In section 2, we review the notion of Lebesgue measure, and we discuss the properties that a "reasonable" measure should have. This is followed by a review of Vitali's theorem in section 3. This is followed by a more introspective consideration of Vitali's proof. In section 4, we consider the key logical arguments that make up Vitali's proof and demonstrate how these have been formalized in ACL2(r). Note that this paper is self-contained. In particular, we do not assume that the reader is intimately familiar with Lebesgue measure, nonstandard analysis, or ACL2(r). Rather, we present the necessary background as it is needed.

2 Lebesgue Measure

The length of an interval of real numbers is the difference of the endpoints of the interval. Intervals include open, closed, and half-open intervals, with their usual meaning: (a, b), $[a, b]$, $(a, b]$, and $[a, b)$. Lebesgue measure extends the notion of length to more complicated sets than intervals.

One way to extend this notion is to introduce some "infinite" reals. The extended reals add the "numbers" $+\infty$ and $-\infty$ to the set of reals. These numbers are introduced so that they generalize the arithmetic and comparison operations in the obvious way. For example, $x < +\infty$ for any x other than $+\infty$. Similarly, $x + \infty = \infty$ for any x other than $-\infty$; the sum $(+\infty) + (-\infty)$ is not defined.

The other way in which the Lebesgue measure $m(S)$ extends the notion of length is to consider the measure of sets S that are not intervals. Lebesgue developed this notion by considering the sum of the length of sets of intervals that completely cover S. The details of that construction are not necessary for the remainder of this paper, but the interested reader is referred to [9].

2.1 Properties of an Ideal Measure

Ideally, the Lebesgue measure m should have the following properties:

1. $m(S)$ is a nonnegative extended real number, for each set of real numbers;
2. for an interval I, $m(I) = \text{length}(I)$;
3. m is countably additive: if $\langle S_n \rangle$ is a sequence of disjoint sets for which m is defined, then $m\left(\bigcup S_n\right) = \sum m(S_n)$;
4. m is translation invariant: if S is a set of reals for which m is defined and r is a real number, let $S + r$ be the set $\{s + r \mid s \in S\}$. Then $m(S + r) = m(S)$.

5. m is finitely additive: if S_1 and S_2 are disjoint sets, for which m is defined, then $m(S_1 \cup S_2) = m(S_1) + m(S_2)$.
6. m is monotonic: if $S_1 \subseteq S_2$ are sets for which m is defined, then $m(S_1) \leq m(S_2)$.

The last two properties can be derived from the previous ones.

Although these properties seem quite reasonable, they are contradictory. For instance, if properties (2)-(4) hold, then using the axiom of choice, a set (called V below) of real numbers can be constructed that cannot have Lesbegue measure, thus property (1) is violated. Such sets are called non-measurable sets.

3 Vitali's Theorem

Given a set S, a σ-algebra is a set of subsets of S that is closed under complements relative to S, finite unions, and countable unions. That is, if $\mathcal{A}$ is a σ-algebra of subsets of the set S, then

$\emptyset \in \mathcal{A}$
if $A \in \mathcal{A}$, then $S - A \in \mathcal{A}$,
if $A \in \mathcal{A}$ and $B \in \mathcal{A}$, then $A \cup B \in \mathcal{A}$, and
if $\langle A_i \rangle$ is a sequence of sets in $\mathcal{A}$, then $\bigcup_{i=0}^{\infty} A_i \in \mathcal{A}$.

By DeMorgan's Laws, a σ-algebra is closed under finite and countable intersections.

In the sequel, let $\mathbb{Q}$ be the set of all rationals and $\mathbb{R}$ be the set of all reals.

Definition 1 (Vitali's Set V). *Let E be the equivalence relation defined by*

$$xEy \Leftrightarrow x, y \in [0,1) \wedge x - y \in \mathbb{Q}.$$

By the Axiom of Choice, there is a set V that contains exactly one element from each equivalence class.

This definition of the set V is essentially due to Vitali [11], who also showed that V is not Lebesgue measurable.

Theorem 1. *If m is a countably additive, translation invariant measure defined on a σ-algebra containing the intervals and the set V (defined above), then $m([0,1))$ is either 0 or infinite.*

Proof. $V \subseteq [0,1)$ has the property that for each $x \in [0,1)$ there is a unique $y \in V$ and a unique $q \in \mathbb{Q}$ such that $x = y + q$.

Consider two cases.

– Case 1. $m(V) = 0$.
Since $[0,1) \subseteq \bigcup\{V + q \,|\, q \in \mathbb{Q}\}$,

$$m([0,1)) \leq m\left(\bigcup\{V + q \,|\, q \in \mathbb{Q}\}\right)$$

$$\begin{aligned} &= \sum_{q \in \mathbb{Q}} m(V + q) \\ &= \sum_{q \in \mathbb{Q}} m(V) \\ &= \sum_{q \in \mathbb{Q}} 0 \\ &= 0. \end{aligned}$$

– Case 2. $m(V) > 0$.
Since

$$\begin{aligned} m\left(\bigcup \{V + q \,|\, 0 \leq q < 1 \wedge q \in \mathbb{Q}\}\right) &= \sum_{q \in [0,1) \cap \mathbb{Q}} m(V + q) \\ &= \sum_{q \in [0,1) \cap \mathbb{Q}} m(V) \\ &= +\infty \end{aligned}$$

and $\bigcup \{V + q \,|\, 0 \leq q < 1 \wedge q \in \mathbb{Q}\} \subseteq [0, 2)$,

$$\begin{aligned} +\infty &= m\left(\bigcup \{V + q \,|\, 0 \leq q < 1 \wedge q \in \mathbb{Q}\}\right) \\ &\leq m([0, 2)). \end{aligned}$$

Thus

$$\begin{aligned} +\infty &= m([0, 2)) \\ &= m([0, 1)) + m([1, 2)) \\ &= m([0, 1)) + m([0, 1) + 1) \\ &= m([0, 1)) + m([0, 1)) \\ &= 2 \cdot m([0, 1)) \end{aligned}$$

and so $m([0, 1)) = +\infty$.

Thus the set V cannot be Lebesque measurable, for otherwise

$$m([0, 1)) \neq 1 = \text{length}([0, 1)).$$

□

We emphasize that we have not developed Lebesgue measure. Peter Loeb[5] found a way to use nonstandard analysis to develop Lebesgue measure on the set of standard real numbers.

4 What Is Needed for the Proof

In order to carry out Vitali's proof, we must have a significant logical machinery in place. First of all, we must have a theory of the real numbers, at least enough

to formalize the reals as a complete, ordered field. Second, we must be able to reason about sets of reals. A complete set theory is not necessary, however. Only enough set theory to construct and manipulate V is required. Finally, the construction of V depends on the Axiom of Choice, so something similar to it must be available. In this section, we show how the logical machinery of ACL2(r) addresses these requirements.

4.1 First-Order Theory of the Reals

ACL2(r) introduces the real numbers using nonstandard analysis. A full treatment of nonstandard analysis can be found in [8], and the formalization of nonstandard analysis in ACL2(r) in [4]. In the following paragraphs, we present a brief description of nonstandard analysis in ACL2(r), for the benefit of readers who are unfamiliar with either.

In nonstandard analysis, the integers are classified as either *standard* or *nonstandard*. All of the familiar integers happen to be *standard*; however, there is at least one nonstandard integer N. Necessarily, $\pm N$, $\pm(N \pm 1)$, $\ldots \pm(N \pm k)$ are all nonstandard for any standard integer k. But notice that if k is nonstandard, $N - k$ may well be standard, e.g., when $k = N$. A number is called *i-large* if its magnitude is larger than any standard integer. The notion of *i-large* captures in a formal sense the intuitive notion of an "infinite" integer. An important fact is that all algebraic properties of the integers hold among both the standard and nonstandard integers.

These notions are easily extended to the reals. There are i-large reals, such as N, $\sqrt{N}$, e^N, etc. Consequently, there are also non-zero reals with magnitude smaller than any standard real, such as $1/N$. Such reals are called *i-small* and correspond with the intuitive notion of "infinitesimal." Note that the only standard number that is also i-small is zero. A number that is not i-large is called *i-limited*. All standard numbers are i-limited, as are all i-small numbers, as is the sum of any two i-limited numbers. Note that the nonstandard integers are precisely the i-large integers.

Two numbers are considered *i-close* if their difference is i-small. We write $x \approx y$ to mean that x is i-close to y. Every i-limited number x is i-close to a standard real, so it can be written as $x = \sigma(x) + \epsilon$, where $\sigma(x)$ is standard and ϵ is i-small. We call $\sigma(x)$ the *standard part* of x. Note that an i-limited number x can be i-close to only *one* standard number, so standard part is well defined. We note that the standard part function takes the place of least upper bounds and completeness of the reals in many arguments in analysis. Specifically, we use this notion for summing infinite series.

While there is much more to nonstandard analysis in ACL2(r), the notions above will be sufficient for the remainder of this paper. Before moving on to other aspects of the proof, however, it is beneficial to address alternative viewpoints of nonstandard analysis. One viewpoint holds that "standard" is a new property of numbers, one that cannot be captured in regular analysis. In this viewpoint,

operations such as + and − are effectively unchanged, and they still operate over the same set of reals as before. An alternative viewpoint is that nonstandard analysis *extends* the real number line by introducing both infinitely small and infinitely large numbers, much in the same way that the reals can be constructed from the rationals.

These two viewpoints can be reconciled. For clarity, we refer to the traditional reals as "the reals" and the extended reals as "the hyperreals". In the first viewpoint, the set $\mathbb{R}$ corresponds to the hyperreals, and the reals are the elements of $\mathbb{R}$ that happen to be standard. In the second viewpoint, the set $\mathbb{R}$ denotes the reals (as usual), which are all standard, and the hyperreals are denoted by $\mathbb{R}^*$. ACL2(r) adopts the first viewpoint, so the predicate `(REALP X)` is true for any hyperreal `X`.

4.2 Sets of Reals

In this section we describe how we represent sets of reals in ACL2(r).

Some sets are represented by designated ACL2(r) λ-expressions. Each such λ-expression, Λ, represents the set of all ACL2(r) objects, a, for which the λ-application $(\Lambda\ a)$ evaluates to an ACL2(r) object other than `NIL`.

For example, the empty set $\emptyset$ is represented by the ACL2(r) λ-expression

```
(LAMBDA (X) (NOT (EQUAL X X))).
```

and the interval $[0, 1)$ is represented by the expression

```
(LAMBDA (X)
       ( (LAMBDA (A B X)
                 (AND (REALP X)
                      (AND (<= A X)
                           (<  X B))))
         '0
         '1
          X )).
```

Note that these are simply ACL2(r) literal constants—not functions. To treat them as functions, they are passed as arguments to an evaluator that mimics their execution. Evaluators can be defined in ACL2 and ACL2(r) for any finite set of previously defined functions. We defined an evaluator that knows about functions such as `and`, `+`, `realp`, `<`, and more specialized functions such as `Vp` and `Seq1` which are useful in the construction of V.

Once the evaluator is defined, it is almost mechanical to define functions to test for set membership, union, intersection, etc. The test for membership calls the evaluator, while the set operations manipulate the λ-expressions. Note that these functions operate over the hyperreals, since ACL2(r) interprets `REALP` as a recognizer for the hyperreals. However, parts of the argument are restricted

to the reals, so we also defined membership predicates and set operations that interpret the expressions as ranging only over the (standard) reals[1].

We mention in passing that the definition of the set equality and subset predicates make use of the ACL2's limited support for quantification. This support is based on the idea of Skolem functions (which find either a witness or a counterexample to model $\exists$ and $\forall$). We will discuss these functions later, in the context of the Axiom of Choice.

We turn our attention now to the definition of countably infinite unions and intersections. Let Φ be a λ-expression, defining a unary function, whose domain includes the standard nonnegative integers, and whose range consists of λ-expressions, like those discussed above, that define sets. Then Φ enumerates a countable collection of sets. Let $\bigcup\Phi$ be the union, over all standard nonnegative integers, k, of the sets represented by the evaluation of the λ-applications, $(\Phi\ k)$.

We can represent $\bigcup\Phi$ in our set notation as follows. First, consider $\bigcup_n \Phi$, defined as the union of the sets $(\Phi\ k)$ for integer k up to n. Then $\bigcup_n \Phi$ can be represented as a λ-expression that encodes a disjunction of the $(\Phi\ k)$. Now suppose that n is an i-large integer. Then $\bigcup_n \Phi$ amounts to a λ-expression containing an "infinite" disjunction, and this is what we use to represent $\bigcup\Phi$. In particular, let $\mathbb{M}$ be the set of standard nonnegative integers. Then ACL2(r) can prove that for standard x, x is a member of $\bigcup\Phi$ if and only if $\exists k \in \mathbb{M}$ such that x is a member of the evaluation of the application $(\Phi\ k)$.

Notice that we don't quite have a σ-algebra of sets: We cannot form "infinite" unions of "infinite" unions. That is "infinite" unions are not allowed to be in the range of the λ-expression Φ. Nevertheless, enough sets exist to carry out the proof.

Since ACL2(r) is first-order, it is not possible in ACL2(r) to quantify over arbitrary σ-algebras and measures. However, we do have the ability to refer explicitly to many sets, including intervals and the set V, via their λ-expression definitions. Moreover, set operations including translations, unions, and even countable unions can all be done by manipulating such definitions.

Recall that ACL2(r) allows constrained functions. So a function m can be consistently constrained to satisfy versions of the required measure axioms that explicitly refer to λ definitions of sets.

For example, m can be constrained to satisfy axioms such as

- If S is the λ definition of a set of (standard) real numbers, then $m(S)$ is a nonnegative extended (standard) real.
- m is finitely and countably additive for definable sets of standard reals.
- m is translation invariant on definable sets of standard reals.

This means that the version of Theorem 1 that we actually prove can be formally stated as follows:

[1] Readers familiar with ACL2(r) may note that evaluators cannot be defined over the function `standardp`, due to limitations regarding recursion in the current version of ACL2(r). Defining different set operations so that `REALP` can be interpreted as a recognizer either for the reals or hyperreals solves this difficulty.

Theorem 2. *If m is a finitely and countably additive, translation invariant measure defined on a σ-algebra containing the λ-definable sets of standard reals, then $m([0,1))$ is either 0 or infinite.*

Since m is constrained, we cannot explicitly state the theorem in ACL2(r), but we can, indeed, carry out the proof!

4.3 The Axiom of Choice

The Axiom of Choice postulates[6]: For every set S of nonempty sets, there is a function f such that for each set $s \in S$, $f(s) \in s$. Such a function f is called a choice function for S.

ACL2(r) implements first-order quantification by axiomatizing Skolem functions. That is, by suitably generalizing the following: $\exists y \varphi(x, y)$ is defined to be $\varphi(x, f(x))$, where x is the free variable in $\exists y \varphi(x, y)$ and the Skolem function f is a new function symbol satisfying the new Skolem axiom $\varphi(x, y) \rightarrow \varphi(x, f(x))$. The Skolem axiom means that the Skolem function can be viewed as a choice function: $f(x)$ chooses a value so that $\varphi(x, f(x)$ will be true, if such a value exists.

ACL2(r) explicitly implements Skolem functions as choice functions. One application of choice functions is the selection of a canonical element from each member of an equivalence class, as is done in Vitali's definition of the set V. However, before ACL2 version 3.1, this was not possible in ACL2.

Some of the discussions at recent ACL2 Workshops centered around this limitation of ACL2, and ACL2 was modified as a result of these discussions. To understand the precise limitation, consider an equivalence relation `E`. The following ACL2 event picks an equivalent `y` for each `x`:

```
(defchoose E-selector-weak (x) (y)
  (E x y))
```

So `(E-selector-weak x)` is always `E`-equivalent to `x`. However, suppose that `x1` is `E`-equivalent to `x2`. Then ACL2 does not guarantee any relationship between `(E-selector-weak x1)` and `(E-selector-weak x2)`. Hence `E-selector-weak` cannot be used to select a canonical member from each `E`-equivalence class.

The solution was to create a stronger `defchoose` function in ACL2, and this is done with the `:strengthen` keyword. In particular, the following `defchoose` *does* select a canonical element from each class:

```
(defchoose E-selector (x) (y)
  (E x y)
  :strengthen t)
```

Now, if `x1` is `E`-equivalent to `x2`, then `(E-selector x1)` is guaranteed to be equal to `(E-selector x2)`. So the range of `E-selector` is the set of canonical elements from the equivalence classes, as needed in the definition of Vitali's set V.

It would be easy to formalize the canonicalizing behavior of `E-selector` in a higher-order logic, but it requires a little care to do so in first-order logic. The `:strengthen` option formalizes this by adding one more constraint on the choice function `E-selector`. In particular, the `defchoose` function introduces the following constraining axioms:

- If there is any x such that $(E\,x\,y)$ is true, then $(E\,(E\text{-}selector\,y)\,y)$ is also true.
- For all possible $y1$, at least one of the following must hold:
  - $(E\text{-}selector\,y) = (E\text{-}selector\,y1)$.
  - $(E\,(E\text{-}selector\,y)\,y)$ is true, but $(E\,(E\text{-}selector\,y)\,y1)$ is false.
  - $(E\,(E\text{-}selector\,y1)\,y1)$ is true, but $(E\,(E\text{-}selector\,y1)\,y)$ is false.

These axioms guarantee that `E-selector` chooses the same canonical value for each y any given equivalence class.

We do not believe that V could have been defined before the introduction of `:strengthen` into ACL2[2]. Specifically, we believe that without `:strengthen`, ACL2 does not have enough logical firepower to carry out many arguments that depend on the Axiom of Choice. However, even without `:strengthen`, ACL2 was able to prove some consequences of the Axiom of Choice that are strictly weaker than the Axiom. One of these is the Principle of Dependent Choices[6].

Definition 2 (Principle of Dependent Choices). *If ρ is a binary relation on a nonempty set S such that for every $x \in S$ there is a $y \in S$ with $x\rho y$, then there is a sequence $\langle x_n \rangle$ of elements from S such that $x_0\rho x_1, x_1\rho x_2, \ldots, x_n\rho x_{n+1}, \ldots$.*

We have proved a version of Dependent Choices in ACL2(r) just using the original Skolem axioms for choice functions, without the strengthening used to establish that choice functions can be made to select unique representatives from equivalence classes[1].

It is noteworthy, however, that Solovay[10] has shown that there is a model of set theory that satisfies the Principle of Dependent Choices, but in which every set of real numbers is Lebesgue measurable.

Moreover, the Principle of Dependent Choices is enough to make it possible to define a satisfactory Lebesgue measure[6]. Dependent Choices ensures, for example, that the set of all real numbers is not the countable union of countable sets and allows proofs of all the "positive" properties, desired by the analysts, of Lebesgue measure.

5 Conclusions

This paper described a formal proof of Vitali's Theorem. The proof depends on three pillars:

- A first-order theory of the reals, as provided by nonstandard analysis in ACL2(r).

[2] In fact, it was precisely this introduction that motivated our current work.

File	Definitions	Theorems	Hints
Analysis fundamentals	17	76	46
Extended reals	10	45	8
Enumeration of rationals	14	64	37
Set Support	50	146	70
Vitali's Construction	3	12	6
Vitali's Proof	0	81	57
Dependent choices	6	4	0

Fig. 1. Effort of work

- A theory of sets sufficient to reason about countable unions of sets of reals. This theory could take many forms, but we chose a representation based on unary λ-expressions and an evaluator that interprets those expressions.
- The Axiom of Choice, which was simulated using Skolem choice functions in ACL2(r).

The third pillar is the most surprising, since it depends on the `:strengthen` feature of Skolem functions, which was only recently introduced into ACL2.

Figure 1 gives an idea of the effort to formalize Vitali's Theorem. The complete ACL2 proof scripts are available from the authors, and they will be added to the ACL2-Books Repository [2].

References

1. Cowles, J.: ACL2 book: Principle of dependent choices. Formal proof script available from author (2010)
2. Davis, J.: Acl2-books repository. Presented at the ACL2 Workshop (2009)
3. Gamboa, R., Cowles, J.: Theory extension in ACL2(r). Journal of Automated Reasoning (May 2007)
4. Gamboa, R., Kaufmann, M.: Nonstandard analysis in ACL2. Journal of Automated Reasoning 27(4), 323–351 (2001)
5. Goldblatt, R.: Lectures on the Hyperreals: An Introduction to Nonstandard Analysis, ch. 16. Springer, Heidelberg (1998)
6. Jech, T.J.: The Axiom of Choice, vol. 75. North-Holland Publishing Company, Amsterdam (1973)
7. Kaufmann, M., Moore, J.S.: Structured theory development for a mechanized logic. Journal of Automated Reasoning 26(2), 161–203 (2001)
8. Nelson, E.: Internal set theory: A new approach to nonstandard analysis. Bulletin of the American Mathematical Society 83, 1165–1198 (1977)
9. Royden, H.L.: Real Analysis, 2nd edn. Macmillan, Basingstoke (1968)
10. Solovay, R.M.: A model of set theory in which every set of reals is Lebesgue measurable. Annals of Mathematics 92, 1–56 (1970)
11. Vitali, G.: Sul problema della mesura dei gruppi di una retta (1905)

A New Foundation for Nominal Isabelle

Brian Huffman[1] and Christian Urban[2]

[1] Portland State University
[2] Technical University of Munich

Abstract. Pitts et al introduced a beautiful theory about names and binding based on the notions of permutation and support. The engineering challenge is to smoothly adapt this theory to a theorem prover environment, in our case Isabelle/HOL. We present a formalisation of this work that differs from our earlier approach in two important respects: First, instead of representing permutations as lists of pairs of atoms, we now use a more abstract representation based on functions. Second, whereas the earlier work modeled different sorts of atoms using different types, we now introduce a unified atom type that includes all sorts of atoms. Interestingly, we allow swappings, that is permutations build from two atoms, to be ill-sorted. As a result of these design changes, we can iron out inconveniences for the user, considerably simplify proofs and also drastically reduce the amount of custom ML-code. Furthermore we can extend the capabilities of Nominal Isabelle to deal with variables that carry additional information. We end up with a pleasing and formalised theory of permutations and support, on which we can build an improved and more powerful version of Nominal Isabelle.

1 Introduction

Nominal Isabelle is a definitional extension of the Isabelle/HOL theorem prover providing a proving infrastructure for convenient reasoning about programming languages. It has been used to formalise an equivalence checking algorithm for LF [11], Typed Scheme [10], several calculi for concurrency [1] and a strong normalisation result for cut-elimination in classical logic [13]. It has also been used by Pollack for formalisations in the locally-nameless approach to binding [9].

At its core Nominal Isabelle is based on the nominal logic work of Pitts et al [5,8]. The most basic notion in this work is a sort-respecting permutation operation defined over a countably infinite collection of sorted atoms. The atoms are used for representing variables that might be bound. Multiple sorts are necessary for being able to represent different kinds of variables. For example, in the language Mini-ML there are bound term variables and bound type variables; each kind needs to be represented by a different sort of atoms.

Unfortunately, the type system of Isabelle/HOL is not a good fit for the way atoms and sorts are used in the original formulation of the nominal logic work. Therefore it was decided in earlier versions of Nominal Isabelle to use a separate type for each sort of atoms and let the type system enforce the sort-respecting property of permutations. Inspired by the work on nominal unification [12], it seemed best at the time to also implement permutations concretely as lists of pairs of atoms. Thus Nominal Isabelle used the two-place permutation operation with the generic type

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 35–50, 2010.

$$\_ \bullet \_ :: (\alpha \times \alpha)\ list \Rightarrow \beta \Rightarrow \beta$$

where α stands for the type of atoms and β for the type of the objects on which the permutation acts. For atoms of type α the permutation operation is defined over the length of lists as follows

$$[] \bullet c \;=\; c \qquad\qquad (a\ b)::\pi \bullet c \;=\; \begin{cases} a & \text{if } \pi \bullet c = b \\ b & \text{if } \pi \bullet c = a \\ \pi \bullet c & \text{otherwise} \end{cases} \tag{1}$$

where we write $(a\ b)$ for a swapping of atoms a and b. For atoms of different type, the permutation operation is defined as $\pi \bullet c \stackrel{def}{=} c$.

With the list representation of permutations it is impossible to state an "ill-sorted" permutation, since the type system excludes lists containing atoms of different type. Another advantage of the list representation is that the basic operations on permutations are already defined in the list library: composition of two permutations (written $\_ \mathbin{@} \_$) is just list append, and inversion of a permutation (written $\_^{-1}$) is just list reversal. A disadvantage is that permutations do not have unique representations as lists; we had to explicitly identify permutations according to the relation

$$\pi_1 \sim \pi_2 \stackrel{def}{=} \forall a.\ \pi_1 \bullet a = \pi_2 \bullet a \tag{2}$$

When lifting the permutation operation to other types, for example sets, functions and so on, we needed to ensure that every definition is well-behaved in the sense that it satisfies the following three *permutation properties*:

$$\begin{array}{ll} \textit{i)} & [] \bullet x = x \\ \textit{ii)} & (\pi_1 \mathbin{@} \pi_2) \bullet x = \pi_1 \bullet (\pi_2 \bullet x) \\ \textit{iii)} & \textit{if } \pi_1 \sim \pi_2 \textit{ then } \pi_1 \bullet x = \pi_2 \bullet x \end{array} \tag{3}$$

From these properties we were able to derive most facts about permutations, and the type classes of Isabelle/HOL allowed us to reason abstractly about these three properties, and then let the type system automatically enforce these properties for each type.

The major problem with Isabelle/HOL's type classes, however, is that they support operations with only a single type parameter and the permutation operations $\_ \bullet \_$ used above in the permutation properties contain two! To work around this obstacle, Nominal Isabelle required the user to declare up-front the collection of *all* atom types, say $\alpha_1,\ldots,\alpha_n$. From this collection it used custom ML-code to generate *n* type classes corresponding to the permutation properties, whereby in these type classes the permutation operation is restricted to

$$\_ \bullet \_ :: (\alpha_i \times \alpha_i)\ list \Rightarrow \beta \Rightarrow \beta$$

This operation has only a single type parameter β (the α_i are the atom types given by the user).

While the representation of permutations-as-lists solved the "sort-respecting" requirement and the declaration of all atom types up-front solved the problem with Isabelle/HOL's type classes, this setup caused several problems for formalising the nominal logic work: First, Nominal Isabelle had to generate n^2 definitions for the permutation operation over n types of atoms. Second, whenever we need to generalise induction hypotheses by quantifying over permutations, we have to build cumbersome quantifications like

$$\forall \pi_1 \ldots \forall \pi_n. \ldots$$

where the π_i are of type $(\alpha_i \times \alpha_i)$ *list*. The reason is that the permutation operation behaves differently for every α_i. Third, although the notion of support

$$supp \_ :: \beta \Rightarrow \alpha \; set$$

which we will define later, has a generic type α *set*, it cannot be used to express the support of an object over *all* atoms. The reason is again that support can behave differently for each α_i. This problem is annoying, because if we need to know in a statement that an object, say x, is finitely supported we end up with having to state premises of the form

$$finite\ ((supp\ x) :: \alpha_1\ set), \ldots, finite\ ((supp\ x) :: \alpha_n\ set) \tag{4}$$

Sometimes we can avoid such premises completely, if x is a member of a *finitely supported type*. However, keeping track of finitely supported types requires another n type classes, and for technical reasons not all types can be shown to be finitely supported.

The real pain of having a separate type for each atom sort arises, however, from another permutation property

$$iv) \quad \pi_1 \bullet (\pi_2 \bullet x) = (\pi_1 \bullet \pi_2) \bullet (\pi_1 \bullet x)$$

where permutation π_1 has type $(\alpha \times \alpha)$ *list*, π_2 type $(\alpha' \times \alpha')$ *list* and x type β. This property is needed in order to derive facts about how permutations of different types interact, which is not covered by the permutation properties *i-iii* shown in (3). The problem is that this property involves three type parameters. In order to use again Isabelle/HOL's type class mechanism with only permitting a single type parameter, we have to instantiate the atom types. Consequently we end up with an additional n^2 slightly different type classes for this permutation property.

While the problems and pain can be almost completely hidden from the user in the existing implementation of Nominal Isabelle, the work is *not* pretty. It requires a large amount of custom ML-code and also forces the user to declare up-front all atom-types that are ever going to be used in a formalisation. In this paper we set out to solve the problems with multiple type parameters in the permutation operation, and in this way can dispense with the large amounts of custom ML-code for generating multiple variants for some basic definitions. The result is that we can implement a pleasingly simple formalisation of the nominal logic work.

Contributions of the paper: Using a single atom type to represent atoms of different sorts and representing permutations as functions are not new ideas. The main

contribution of this paper is to show an example of how to make better theorem proving tools by choosing the right level of abstraction for the underlying theory—our design choices take advantage of Isabelle's type system, type classes, and reasoning infrastructure. The novel technical contribution is a mechanism for dealing with "Church-style" lambda-terms [4] and HOL-based languages [7] where variables and variable binding depend on type annotations.

2 Sorted Atoms and Sort-Respecting Permutations

In the nominal logic work of Pitts, binders and bound variables are represented by *atoms*. As stated above, we need to have different *sorts* of atoms to be able to bind different kinds of variables. A basic requirement is that there must be a countably infinite number of atoms of each sort. Unlike in our earlier work, where we identified each sort with a separate type, we implement here atoms to be

datatype $atom = Atom\ string\ nat$

whereby the string argument specifies the sort of the atom.[1] (The use type *string* is merely for convenience; any countably infinite type would work as well.) We have an auxiliary function *sort* that is defined as $sort\ (Atom\ s\ i) = s$, and we clearly have for every finite set X of atoms and every sort s the property:

Proposition 1. *If finite X then there exists an atom a such that* $sort\ a = s$ *and* $a \notin X$.

For implementing sort-respecting permutations, we use functions of type $atom \Rightarrow atom$ that *i*) are bijective; *ii*) are the identity on all atoms, except a finite number of them; and *iii*) map each atom to one of the same sort. These properties can be conveniently stated for a function π as follows:

$$i)\ \ bij\ \pi \qquad ii)\ \ finite\ \{a \mid \pi\ a \neq a\} \qquad iii)\ \ \forall a.\ sort\ (\pi\ a) = sort\ a \tag{5}$$

Like all HOL-based theorem provers, Isabelle/HOL allows us to introduce a new type *perm* that includes just those functions satisfying all three properties. For example the identity function, written *id*, is included in *perm*. Also function composition, written $\_ \circ \_$, and function inversion, given by Isabelle/HOL's inverse operator and written $inv\ \_$, preserve the properties *i-iii*.

However, a moment of thought is needed about how to construct non-trivial permutations. In the nominal logic work it turned out to be most convenient to work with swappings, written $(a\ b)$. In our setting the type of swappings must be

$$(\_\ \_) :: atom \Rightarrow atom \Rightarrow perm$$

but since permutations are required to respect sorts, we must carefully consider what happens if a user states a swapping of atoms with different sorts. In earlier versions of Nominal Isabelle, we avoided this problem by using different types for different

[1] A similar design choice was made by Gunter et al [6] for their variables.

sorts; the type system prevented users from stating ill-sorted swappings. Here, however, definitions such as[2]

$$(a\ b) \stackrel{def}{=} \lambda c.\ \mathit{if}\ a = c\ \mathit{then}\ b\ \mathit{else}\ (\mathit{if}\ b = c\ \mathit{then}\ a\ \mathit{else}\ c)$$

do not work in general, because the type system does not prevent a and b from having different sorts—in which case the function would violate property *iii*. We could make the definition of swappings partial by adding the precondition $\mathit{sort}\ a = \mathit{sort}\ b$, which would mean that in case a and b have different sorts, the value of $(a\ b)$ is unspecified. However, this looked like a cumbersome solution, since sort-related side conditions would be required everywhere, even to unfold the definition. It turned out to be more convenient to actually allow the user to state "ill-sorted" swappings but limit their "damage" by defaulting to the identity permutation in the ill-sorted case:

$$\begin{array}{l}(a\ b) \stackrel{def}{=} \mathit{if}\ (\mathit{sort}\ a = \mathit{sort}\ b)\\ \qquad\quad \mathit{then}\ \lambda c.\ \mathit{if}\ a = c\ \mathit{then}\ b\ \mathit{else}\ (\mathit{if}\ b = c\ \mathit{then}\ a\ \mathit{else}\ c)\\ \qquad\quad \mathit{else}\ \mathit{id}\end{array} \tag{6}$$

This function is bijective, the identity on all atoms except a and b, and sort respecting. Therefore it is a function in *perm*.

One advantage of using functions instead of lists as a representation for permutations is that for example the swappings

$$(a\ b) = (b\ a) \qquad (a\ a) = \mathit{id} \tag{7}$$

are *equal*. We do not have to use the equivalence relation shown in (2) to identify them, as we would if they had been represented as lists of pairs. Another advantage of the function representation is that they form a (non-commutative) group, provided we define

$$0 \stackrel{def}{=} \mathit{id} \qquad \pi_1 + \pi_2 \stackrel{def}{=} \pi_1 \circ \pi_2 \qquad -\pi \stackrel{def}{=} \mathit{inv}\ \pi \qquad \pi_1 - \pi_2 \stackrel{def}{=} \pi_1 + -\pi_2$$

and verify the simple properties

$$\pi_1 + \pi_2 + \pi_3 = \pi_1 + (\pi_2 + \pi_3) \qquad 0 + \pi = \pi \qquad \pi + 0 = \pi \qquad -\pi + \pi = 0$$

Again this is in contrast to the list-of-pairs representation which does not form a group. The technical importance of this fact is that we can rely on Isabelle/HOL's existing simplification infrastructure for groups, which will come in handy when we have to do calculations with permutations. Note that Isabelle/HOL defies standard conventions of mathematical notation by using additive syntax even for non-commutative groups. Obviously, composition of permutations is not commutative in general—$\pi_1 + \pi_2 \neq \pi_2 + \pi_1$. But since the point of this paper is to implement the nominal theory as smoothly as possible in Isabelle/HOL, we tolerate the non-standard notation in order to reuse the existing libraries.

By formalising permutations abstractly as functions, and using a single type for all atoms, we can now restate the *permutation properties* from (3) as just the two equations

[2] To increase legibility, we omit here and in what follows the *Rep_perm* and *Abs_perm* wrappers that are needed in our implementation since we defined permutation not to be the full function space, but only those functions of type *perm* satisfying properties *i-iii*.

$$\begin{array}{ll} i) & 0 \bullet x = x \\ ii) & (\pi_1 + \pi_2) \bullet x = \pi_1 \bullet \pi_2 \bullet x \end{array} \tag{8}$$

in which the permutation operations are of type $perm \Rightarrow \beta \Rightarrow \beta$ and so have only a single type parameter. Consequently, these properties are compatible with the one-parameter restriction of Isabelle/HOL's type classes. There is no need to introduce a separate type class instantiated for each sort, like in the old approach.

The next notion allows us to establish generic lemmas involving the permutation operation.

Definition 1. *A type β is a* permutation type *if the permutation properties in (8) are satisfied for every x of type β.*

First, it follows from the laws governing groups that a permutation and its inverse cancel each other. That is, for any x of a permutation type:

$$\pi \bullet (-\pi) \bullet x = x \qquad (-\pi) \bullet \pi \bullet x = x \tag{9}$$

Consequently, in a permutation type the permutation operation $\pi \bullet \_$ is bijective, which in turn implies the property

$$\pi \bullet x = \pi \bullet y \;\; \textit{if and only if} \;\; x = y. \tag{10}$$

In order to lift the permutation operation to other types, we can define for:

$$\begin{array}{rl} \textit{atoms:} & \pi \bullet a \stackrel{def}{=} \pi\, a \\ \textit{functions:} & \pi \bullet f \stackrel{def}{=} \lambda x.\ \pi \bullet (f\ ((-\pi) \bullet x)) \\ \textit{permutations:} & \pi \bullet \pi' \stackrel{def}{=} \pi + \pi' - \pi \\ \textit{sets:} & \pi \bullet X \stackrel{def}{=} \{\pi \bullet x \mid x \in X\} \\ \textit{booleans:} & \pi \bullet b \stackrel{def}{=} b \end{array} \qquad \begin{array}{rl} \textit{lists:} & \pi \bullet [] \stackrel{def}{=} [] \\ & \pi \bullet (x::xs) \stackrel{def}{=} (\pi \bullet x)::(\pi \bullet xs) \\ \textit{products:} & \pi \bullet (x, y) \stackrel{def}{=} (\pi \bullet x, \pi \bullet y) \\ \textit{nats:} & \pi \bullet n \stackrel{def}{=} n \end{array}$$

and then establish:

Theorem 1. *If β, β_1 and β_2 are permutation types, then so are atom, $\beta_1 \Rightarrow \beta_2$, perm, β set, β list, $\beta_1 \times \beta_2$, bool and nat.*

Proof. All statements are by unfolding the definitions of the permutation operations and simple calculations involving addition and minus. With permutations for example we have

$$\begin{aligned} 0 \bullet \pi' &\stackrel{def}{=} 0 + \pi' - 0 = \pi' \\ (\pi_1 + \pi_2) \bullet \pi' &\stackrel{def}{=} (\pi_1 + \pi_2) + \pi' - (\pi_1 + \pi_2) \\ &= (\pi_1 + \pi_2) + \pi' - \pi_2 - \pi_1 \\ &= \pi_1 + (\pi_2 + \pi' - \pi_2) - \pi_1 \stackrel{def}{=} \pi_1 \bullet \pi_2 \bullet \pi' \end{aligned}$$ □

The main point is that the above reasoning blends smoothly with the reasoning infrastructure of Isabelle/HOL; no custom ML-code is necessary and a single type class suffices. We can also show once and for all that the following property—which caused so many headaches in our earlier setup—holds for any permutation type.

Lemma 1. *Given x is of permutation type, then* $\pi_1 \bullet (\pi_2 \bullet x) = (\pi_1 \bullet \pi_2) \bullet (\pi_1 \bullet x)$.

Proof. The proof is as follows:

$$\begin{array}{lll} \pi_1 \bullet \pi_2 \bullet x & = & \pi_1 \bullet \pi_2 \bullet (-\pi_1) \bullet \pi_1 \bullet x \qquad \textit{by (9)} \\ & = & (\pi_1 + \pi_2 - \pi_1) \bullet \pi_1 \bullet x \qquad \textit{by (8.ii)} \\ & \stackrel{def}{=} & (\pi_1 \bullet \pi_2) \bullet (\pi_1 \bullet x) \end{array}$$

□

An *equivariant* function or predicate is one that is invariant under the swapping of atoms. Having a notion of equivariance with nice logical properties is a major advantage of bijective permutations over traditional renaming substitutions [8, §2]. Equivariance can be defined uniformly for all permutation types, and it is satisfied by most HOL functions and constants.

Definition 2. *A function f is* equivariant *if* $\forall \pi.\ \pi \bullet f = f$.

There are a number of equivalent formulations for the equivariance property. For example, assuming f is of type $\alpha \Rightarrow \beta$, then equivariance can also be stated as

$$\forall \pi\, x.\ \ \pi \bullet (f\, x) = f\ (\pi \bullet x) \tag{11}$$

To see that this formulation implies the definition, we just unfold the definition of the permutation operation for functions and simplify with the equation and the cancellation property shown in (9). To see the other direction, we use the fact

$$\pi \bullet (f\, x) = (\pi \bullet f)\ (\pi \bullet x) \tag{12}$$

which follows again directly from the definition of the permutation operation for functions and the cancellation property. Similarly for functions with more than one argument.

Both formulations of equivariance have their advantages and disadvantages: (11) is often easier to establish. For example we can easily show that equality is equivariant

$$\pi \bullet (x = y) = (\pi \bullet x = \pi \bullet y)$$

using the permutation operation on booleans and property (10). Lemma 1 establishes that the permutation operation is equivariant. It is also easy to see that the boolean operators, like $\wedge$, $\vee$ and $\longrightarrow$ are all equivariant. Furthermore a simple calculation will show that our swapping functions are equivariant, that is

$$\pi \bullet (a\ b) = ((\pi \bullet a)\ (\pi \bullet b)) \tag{13}$$

for all a, b and π. These equivariance properties are tremendously helpful later on when we have to push permutations inside terms.

3 Support and Freshness

The most original aspect of the nominal logic work of Pitts et al is a general definition for "the set of free variables of an object x". This definition is general in the sense that it applies not only to lambda-terms, but also to lists, products, sets and even functions. The definition depends only on the permutation operation and on the notion of equality defined for the type of x, namely:

$$supp\ x \stackrel{def}{=} \{a \mid infinite\ \{b \mid (a\ b) \bullet x \neq x\}\}$$

(Note that due to the definition of swapping in (6), we do not need to explicitly restrict a and b to have the same sort.) There is also the derived notion for when an atom a is *fresh* for an x, defined as

$$a \# x \stackrel{def}{=} a \notin supp\ x$$

A striking consequence of these definitions is that we can prove without knowing anything about the structure of x that swapping two fresh atoms, say a and b, leave x unchanged. For the proof we use the following lemma about swappings applied to an x:

Lemma 2. *Assuming x is of permutation type, and a, b and c have the same sort, then $(a\ c) \bullet x = x$ and $(b\ c) \bullet x = x$ imply $(a\ b) \bullet x = x$.*

Proof. The cases where $a = c$ and $b = c$ are immediate. For the remaining case it is, given our assumptions, easy to calculate that the permutations

$$(a\ c) + (b\ c) + (a\ c) = (a\ b)$$

are equal. The lemma is then by application of the second permutation property shown in (8). □

Theorem 2. *Let x be of permutation type. If $a \# x$ and $b \# x$ then $(a\ b) \bullet x = x$.*

Proof. If a and b have different sort, then the swapping is the identity. If they have the same sort, we know by definition of support that both $finite\ \{c \mid (a\ c) \bullet x \neq x\}$ and $finite\ \{c \mid (b\ c) \bullet x \neq x\}$ hold. So the union of these sets is finite too, and we know by Proposition 1 that there is an atom c, with the same sort as a and b, that satisfies $(a\ c) \bullet x = x$ and $(b\ c) \bullet x = x$. Now the theorem follows from Lemma 2. □

Two important properties that need to be established for later calculations is that *supp* and freshness are equivariant. For this we first show that:

Lemma 3. *If x is a permutation type, then $\pi \bullet a \# \pi \bullet x$ if and only if $a \# x$.*

Proof.

$$\begin{array}{lll} & \pi \bullet a \# \pi \bullet x \stackrel{def}{=} finite\ \{b \mid ((\pi \bullet a)\ b) \bullet \pi \bullet x \neq \pi \bullet x\} & \\ \Leftrightarrow & finite\ \{b \mid ((\pi \bullet a)\ (\pi \bullet b)) \bullet \pi \bullet x \neq \pi \bullet x\} & since\ \pi \bullet \_\ is\ bijective \\ \Leftrightarrow & finite\ \{b \mid \pi \bullet (a\ b) \bullet x \neq \pi \bullet x\} & by\ (11)\ and\ (13) \\ \Leftrightarrow & finite\ \{b \mid (a\ b) \bullet x \neq x\} \stackrel{def}{=} a \# x & by\ (10) \end{array}$$

□

Together with the definition of the permutation operation on booleans, we can immediately infer equivariance of freshness:

$$\pi \bullet (a \# x) = (\pi \bullet a \# \pi \bullet x)$$

Now equivariance of *supp*, namely

$$\pi \bullet (supp\ x) = supp\ (\pi \bullet x)$$

is by noting that $supp\ x = \{a \mid \neg\ a \# x\}$ and that freshness and the logical connectives are equivariant.

While the abstract properties of support and freshness, particularly Theorem 2, are useful for developing Nominal Isabelle, one often has to calculate the support of some concrete object. This is straightforward for example for booleans, nats, products and lists:

$$\begin{array}{rl} \textit{booleans}: & supp\ b = \varnothing \\ \textit{nats}: & supp\ n = \varnothing \\ \textit{products}: & supp\ (x, y) = supp\ x \cup supp\ y \end{array} \qquad \begin{array}{rl} \textit{lists}: & supp\ [] = \varnothing \\ & supp\ (x{::}xs) = supp\ x \cup supp\ xs \end{array}$$

But establishing the support of atoms and permutations in our setup here is a bit trickier. To do so we will use the following notion about a *supporting set*.

Definition 3. *A set S* supports *x if for all atoms a and b not in S we have* $(a\ b) \bullet x = x$.

The main motivation for this notion is that we can characterise *supp x* as the smallest finite set that supports *x*. For this we prove:

Lemma 4. *Let x be of permutation type.*

i) If S supports x and finite S then supp $x \subseteq S$.
ii) $(supp\ x)$ *supports x*
iii) supp $x = S$ *provided S supports x, finite S and S is the least such set, that means formally, for all* S'*, if finite* S' *and* S' *supports x then* $S \subseteq S'$.

Proof. For *i*) we derive a contradiction by assuming there is an atom *a* with $a \in supp\ x$ and $a \notin S$. Using the second fact, the assumption that *S supports x* gives us that *S* is a superset of $\{b \mid (a\ b) \bullet x \neq x\}$, which is finite by the assumption of *S* being finite. But this means $a \notin supp\ x$, contradicting our assumption. Property *ii*) is by a direct application of Theorem 2. For the last property, part *i*) proves one "half" of the claimed equation. The other "half" is by property *ii*) and the fact that *supp x* is finite by *i*). □

These are all relatively straightforward proofs adapted from the existing nominal logic work. However for establishing the support of atoms and permutations we found the following "optimised" variant of *iii*) more useful:

Lemma 5. *Let x be of permutation type. We have that* $supp\ x = S$ *provided S supports x, finite S, and for all* $a \in S$ *and all* $b \notin S$*, with a and b having the same sort,* $(a\ b) \bullet x \neq x$

Proof. By Lemma 4.*iii*) we have to show that for every finite set S' that supports x, $S \subseteq S'$ holds. We will assume that there is an atom a that is element of S, but not S' and derive a contradiction. Since both S and S' are finite, we can by Proposition 1 obtain an atom b, which has the same sort as a and for which we know $b \notin S$ and $b \notin S'$. Since we assumed $a \notin S'$ and we have that S' *supports* x, we have on one hand $(a\ b) \bullet x = x$. On the other hand, the fact $a \in S$ and $b \notin S$ imply $(a\ b) \bullet x \neq x$ using the assumed implication. This gives us the contradiction. □

Using this lemma we only have to show the following three proof-obligations

$$\begin{array}{ll} i) & \{c\} \textit{ supports } c \\ ii) & \textit{finite } \{c\} \\ iii) & \forall a \in \{c\}\ b \notin \{c\}.\ \textit{sort } a = \textit{sort } b \longrightarrow (a\ b) \bullet c \neq c \end{array}$$

in order to establish that $\textit{supp}\ c = \{c\}$ holds. In Isabelle/HOL these proof-obligations can be discharged by easy simplifications. Similar proof-obligations arise for the support of permutations, which is

$$\textit{supp}\ \pi = \{a \mid \pi \bullet a \neq a\}$$

The only proof-obligation that is interesting is the one where we have to show that

$$\textit{If } \pi \bullet a \neq a, \pi \bullet b = b \textit{ and sort } a = \textit{sort } b, \textit{ then } (a\ b) \bullet \pi \neq \pi.$$

For this we observe that

$$(a\ b) \bullet \pi = \pi \textit{ if and only if } \pi \bullet (a\ b) = (a\ b)$$

holds by a simple calculation using the group properties of permutations. The proof-obligation can then be discharged by analysing the inequality between the permutations $((\pi \bullet a)\ b)$ and $(a\ b)$.

The main point about support is that whenever an object x has finite support, then Proposition 1 allows us to choose for x a fresh atom with arbitrary sort. This is an important operation in Nominal Isabelle in situations where, for example, a bound variable needs to be renamed. To allow such a choice, we only have to assume *one* premise of the form $\textit{finite}\ (\textit{supp}\ x)$ for each x. Compare that with the sequence of premises in our earlier version of Nominal Isabelle (see (4)). For more convenience we can define a type class for types where every element has finite support, and prove that the types *atom*, *perm*, lists, products and booleans are instances of this type class. Then *no* premise is needed, as the type system of Isabelle/HOL can figure out automatically when an object is finitely supported.

Unfortunately, this does not work for sets or Isabelle/HOL's function type. There are functions and sets definable in Isabelle/HOL for which the finite support property does not hold. A simple example of a function with infinite support is the function that returns the natural number of an atom

$$\textit{nat\_of}\ (\textit{Atom}\ s\ i) \stackrel{\textit{def}}{=} i$$

This function's support is the set of *all* atoms. To establish this we show $\neg\, a \,\#\, nat\_of$. This is equivalent to assuming the set $\{b \mid (a\ b) \bullet nat\_of \neq nat\_of\}$ is finite and deriving a contradiction. From the assumption we also know that $\{a\} \cup \{b \mid (a\ b) \bullet nat\_of \neq nat\_of\}$ is finite. Then we can use Proposition 1 to choose an atom c such that $c \neq a$, $sort\ c = sort\ a$ and $(a\ c) \bullet nat\_of = nat\_of$. Now we can reason as follows:

$$\begin{array}{lll} nat\_of\ a = (a\ \ c) \bullet (nat\_of\ a) & & \textit{by def. of permutations on nats} \\ \quad = ((a\ c) \bullet nat\_of)\ ((a\ c) \bullet a) & & \textit{by (12)} \\ \quad = nat\_of\ c & & \textit{by assumptions on } c \end{array}$$

But this means we have that $nat\_of\ a = nat\_of\ c$ and $sort\ a = sort\ c$. This implies that atoms a and c must be equal, which clashes with our assumption $c \neq a$ about how we chose c. Cheney [3] gives similar examples for constructions that have infinite support.

4 Concrete Atom Types

So far, we have presented a system that uses only a single multi-sorted atom type. This design gives us the flexibility to define operations and prove theorems that are generic with respect to atom sorts. For example, as illustrated above the *supp* function returns a set that includes the free atoms of *all* sorts together; the flexibility offered by the new atom type makes this possible.

However, the single multi-sorted atom type does not make an ideal interface for end-users of Nominal Isabelle. If sorts are not distinguished by Isabelle's type system, users must reason about atom sorts manually. That means subgoals involving sorts must be discharged explicitly within proof scripts, instead of being inferred by Isabelle/HOL's type checker. In other cases, lemmas might require additional side conditions about sorts to be true. For example, swapping a and b in the pair (a, b) will only produce the expected result if we state the lemma in Isabelle/HOL as:

lemma
  fixes $a\ b :: atom$
  assumes asm: $sort\ a = sort\ b$
  shows $(a\ \ b) \bullet (a, b) = (b, a)$
using asm **by** $simp$

Fortunately, it is possible to regain most of the type-checking automation that is lost by moving to a single atom type. We accomplish this by defining *subtypes* of the generic atom type that only include atoms of a single specific sort. We call such subtypes *concrete atom types*.

The following Isabelle/HOL command defines a concrete atom type called *name*, which consists of atoms whose sort equals the string $''name''$.

typedef $name = \{a \mid sort\ a = ''name''\}$

This command automatically generates injective functions that map from the concrete atom type into the generic atom type and back, called representation and abstraction functions, respectively. We will write these functions as follows:

$$\lfloor \_ \rfloor :: name \Rightarrow atom \qquad \lceil \_ \rceil :: atom \Rightarrow name$$

With the definition $\pi \bullet a \stackrel{def}{=} \lceil \pi \bullet \lfloor a \rfloor \rceil$, it is straightforward to verify that the type *name* is a permutation type.

In order to reason uniformly about arbitrary concrete atom types, we define a type class that characterises type *name* and other similarly-defined types. The definition of the concrete atom type class is as follows: First, every concrete atom type must be a permutation type. In addition, the class defines an overloaded function that maps from the concrete type into the generic atom type, which we will write $|\_|$. For each class instance, this function must be injective and equivariant, and its outputs must all have the same sort, that is

$$i)\ \textit{if}\ |a| = |b|\ \textit{then}\ a = b \qquad ii)\ \pi \bullet |a| = |\pi \bullet a| \qquad iii)\ \textit{sort}\ |a| = \textit{sort}\ |b| \tag{14}$$

With the definition $|a| \stackrel{def}{=} \lfloor a \rfloor$ we can show that *name* satisfies all the above requirements of a concrete atom type.

The whole point of defining the concrete atom type class was to let users avoid explicit reasoning about sorts. This benefit is realised by defining a special swapping operation of type $\alpha \Rightarrow \alpha \Rightarrow perm$, where α is a concrete atom type:

$$(a \leftrightarrow b) \stackrel{def}{=} (|a|\ |b|)$$

As a consequence of its type, the $\leftrightarrow$-swapping operation works just like the generic swapping operation, but it does not require any sort-checking side conditions—the sort-correctness is ensured by the types! For $\leftrightarrow$ we can establish the following simplification rule:

$$(a \leftrightarrow b) \bullet c = (\textit{if}\ c = a\ \textit{then}\ b\ \textit{else if}\ c = b\ \textit{then}\ a\ \textit{else}\ c)$$

If we now want to swap the *concrete* atoms a and b in the pair (a, b) we can establish the lemma as follows:

lemma
fixes $a\ b :: name$
shows $(a \leftrightarrow b) \bullet (a, b) = (b, a)$
by *simp*

There is no need to state an explicit premise involving sorts.

We can automate the process of creating concrete atom types, so that users can define a new one simply by issuing the command

atom_decl *name*

This command can be implemented using less than 100 lines of custom ML-code. In comparison, the old version of Nominal Isabelle included more than 1000 lines of ML-code for creating concrete atom types, and for defining various type classes and instantiating generic lemmas for them. In addition to simplifying the ML-code, the setup here also offers user-visible improvements: Now concrete atoms can be declared at any point of a formalisation, and theories that separately declare different atom types can be merged together—it is no longer required to collect all atom declarations in one place.

5 Multi-sorted Concrete Atoms

The formalisation presented so far allows us to streamline proofs and reduce the amount of custom ML-code in the existing implementation of Nominal Isabelle. In this section we describe a mechanism that extends the capabilities of Nominal Isabelle. This mechanism is about variables with additional information, for example typing constraints. While we leave a detailed treatment of binders and binding of variables for a later paper, we will have a look here at how such variables can be represented by concrete atoms.

In the previous section we considered concrete atoms that can be used in simple binders like $\lambda x.\ x$. Such concrete atoms do not carry any information beyond their identities—comparing for equality is really the only way to analyse ordinary concrete atoms. However, in "Church-style" lambda-terms [4] and in the terms underlying HOL-systems [7] binders and bound variables have a more complicated structure. For example in the "Church-style" lambda-term

$$\lambda x_\alpha.\ x_\alpha\ x_\beta \tag{15}$$

both variables and binders include typing information indicated by α and β. In this setting, we treat x_α and x_β as distinct variables (assuming $\alpha \neq \beta$) so that the variable x_α is bound by the lambda-abstraction, but not x_β.

To illustrate how we can deal with this phenomenon, let us represent object types like α and β by the datatype

datatype *ty* = *TVar string* | *ty* → *ty*

If we attempt to encode a variable naively as a pair of a *name* and a *ty*, we have the problem that a swapping, say $(x \leftrightarrow y)$, applied to the pair $((x, \alpha), (x, \beta))$ will always permute *both* occurrences of x, even if the types α and β are different. This is unwanted, because it will eventually mean that both occurrences of x will become bound by a corresponding binder.

Another attempt might be to define variables as an instance of the concrete atom type class, where a *ty* is somehow encoded within each variable. Remember we defined atoms as the datatype:

datatype *atom* = *Atom string nat*

Considering our method of defining concrete atom types, the usage of a string for the sort of atoms seems a natural choice. However, none of the results so far depend on this choice and we are free to change it. One possibility is to encode types or any other information by making the sort argument parametric as follows:

datatype *'a atom* = *Atom 'a nat*

The problem with this possibility is that we are then back in the old situation where our permutation operation is parametric in two types and this would require to work around Isabelle/HOL's restriction on type classes. Fortunately, encoding the types in a separate parameter is not necessary for what we want to achieve, as we only have to know when two types are equal or not. The solution is to use a different sort for each

object type. Then we can use the fact that permutations respect *sorts* to ensure that permutations also respect *object types*. In order to do this, we must define an injective function *sort_ty* mapping from object types to sorts. For defining functions like *sort_ty*, it is more convenient to use a tree datatype for sorts. Therefore we define

datatype $sort = Sort\ string\ (sort\ list)$
datatype $atom = Atom\ sort\ nat$

With this definition, the sorts we considered so far can be encoded just as $Sort\ s\ []$. The point, however, is that we can now define the function *sort_ty* simply as

$$\begin{array}{l} sort\_ty\ (TVar\ s) = Sort\ ''TVar''\ [Sort\ s\ []] \\ sort\_ty\ (\tau_1 \rightarrow \tau_2) = Sort\ ''Fun''\ [sort\_ty\ \tau_1, sort\_ty\ \tau_2] \end{array} \tag{16}$$

which can easily be shown to be injective.

Having settled on what the sorts should be for "Church-like" atoms, we have to give a subtype definition for concrete atoms. Previously we identified a subtype consisting of atoms of only one specified sort. This must be generalised to all sorts the function *sort_ty* might produce, i.e. the range of *sort_ty*. Therefore we define

typedef $var = \{a \mid sort\ a \in range\ sort\_ty\}$

This command gives us again injective representation and abstraction functions. We will write them also as $\lfloor \_ \rfloor :: var \Rightarrow atom$ and $\lceil \_ \rceil :: atom \Rightarrow var$, respectively.

We can define the permutation operation for *var* as $\pi \bullet a \stackrel{def}{=} \lceil \pi \bullet \lfloor a \rfloor \rceil$ and the injective function to type *atom* as $|a| \stackrel{def}{=} \lfloor a \rfloor$. Finally, we can define a constructor function that makes a *var* from a variable name and an object type:

$$Var\ x\ \alpha \stackrel{def}{=} \lceil Atom\ (sort\_ty\ \alpha)\ x \rceil$$

With these definitions we can verify all the properties for concrete atom types except Property 14.*iii*), which requires every atom to have the same sort. This last property is clearly not true for type *var*. This fact is slightly unfortunate since this property allowed us to use the type-checker in order to shield the user from all sort-constraints. But this failure is expected here, because we cannot burden the type-system of Isabelle/HOL with the task of deciding when two object types are equal. This means we sometimes need to explicitly state sort constraints or explicitly discharge them, but as we will see in the lemma below this seems a natural price to pay in these circumstances.

To sum up this section, the encoding of type-information into atoms allows us to form the swapping $(Var\ x\ \alpha \leftrightarrow Var\ y\ \alpha)$ and to prove the following lemma

lemma
assumes *asm*: $\alpha \neq \beta$
shows $(Var\ x\ \alpha \leftrightarrow Var\ y\ \alpha) \bullet (Var\ x\ \alpha, Var\ x\ \beta) = (Var\ y\ \alpha, Var\ x\ \beta)$
using *asm* **by** *simp*

As we expect, the atom $Var\ x\ \beta$ is left unchanged by the swapping. With this we can faithfully represent bindings in languages involving "Church-style" terms and bindings

as shown in (15). We expect that the creation of such atoms can be easily automated so that the user just needs to specify **atom_decl** *var* (*ty*) where the argument, or arguments, are datatypes for which we can automatically define an injective function like *sort_ty* (see (16)). Our hope is that with this approach Benzmueller and Paulson can make headway with formalising their results about simple type theory [2]. Because of its limitations, they did not attempt this with the old version of Nominal Isabelle. We also hope we can make progress with formalisations of HOL-based languages.

6 Conclusion

This proof pearl describes a new formalisation of the nominal logic work by Pitts et al. With the definitions we presented here, the formal reasoning blends smoothly with the infrastructure of the Isabelle/HOL theorem prover. Therefore the formalisation will be the underlying theory for a new version of Nominal Isabelle.

The main difference of this paper with respect to existing work on Nominal Isabelle is the representation of atoms and permutations. First, we used a single type for sorted atoms. This design choice means for a term t, say, that its support is completely characterised by *supp* t, even if the term contains different kinds of atoms. Also, whenever we have to generalise an induction so that a property P is not just established for all t, but for all t *and* under all permutations π, then we only have to state $\forall \pi.\ P\ (\pi \bullet t)$. The reason is that permutations can now consist of multiple swapping each of which can swap different kinds of atoms. This simplifies considerably the reasoning involved in building Nominal Isabelle.

Second, we represented permutations as functions so that the associated permutation operation has only a single type parameter. This is very convenient because the abstract reasoning about permutations fits cleanly with Isabelle/HOL's type classes. No custom ML-code is required to work around rough edges. Moreover, by establishing that our permutations-as-functions representation satisfy the group properties, we were able to use extensively Isabelle/HOL's reasoning infrastructure for groups. This often reduced proofs to simple calculations over $+$, $-$ and 0. An interesting point is that we defined the swapping operation so that a swapping of two atoms with different sorts is *not* excluded, like in our older work on Nominal Isabelle, but there is no "effect" of such a swapping (it is defined as the identity). This is a crucial insight in order to make the approach based on a single type of sorted atoms to work. But of course it is analogous to the well-known trick of defining division by zero to return zero.

We noticed only one disadvantage of the permutations-as-functions: Over lists we can easily perform inductions. For permutations made up from functions, we have to manually derive an appropriate induction principle. We can establish such a principle, but we have no real experience yet whether ours is the most useful principle: such an induction principle was not needed in any of the reasoning we ported from the old Nominal Isabelle, except when showing that if $\forall a \in supp\ x.\ a \# p$ implies $p \bullet x = x$.

Finally, our implementation of sorted atoms turned out powerful enough to use it for representing variables that carry on additional information, for example typing annotations. This information is encoded into the sorts. With this we can represent conveniently binding in "Church-style" lambda-terms and HOL-based languages. While

dealing with such additional information in dependent type-theories, such as LF or Coq, is straightforward, we are not aware of any other approach in a non-dependent HOL-setting that can deal conveniently with such binders.

The formalisation presented here will eventually become part of the Isabelle distribution, but for the moment it can be downloaded from the Mercurial repository linked at http://isabelle.in.tum.de/nominal/download

Acknowledgements. We are very grateful to Jesper Bengtson, Stefan Berghofer and Cezary Kaliszyk for their comments on earlier versions of this paper. We are also grateful to the anonymous referee who helped us to put the work into the right context.

References

1. Bengtson, J., Parrow, J.: Formalising the pi-Calculus using Nominal Logic. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 63–77. Springer, Heidelberg (2007)
2. Benzmüller, C., Paulson, L.C.: Quantified Multimodal Logics in Simple Type Theory. SEKI Report SR–2009–02 (ISSN 1437-4447). SEKI Publications (2009), http://arxiv.org/abs/0905.2435
3. Cheney, J.: Completeness and Herbrand Theorems for Nominal Logic. Journal of Symbolic Logic 71(1), 299–320 (2006)
4. Church, A.: A Formulation of the Simple Theory of Types. Journal of Symbolic Logic 5(2), 56–68 (1940)
5. Gabbay, M.J., Pitts, A.M.: A New Approach to Abstract Syntax with Variable Binding. Formal Aspects of Computing 13, 341–363 (2002)
6. Gunter, E., Osborn, C., Popescu, A.: Theory Support for Weak Higher Order Abstract Syntax in Isabelle/HOL. In: Proc. of the 4th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP). ENTCS, pp. 12–20 (2009)
7. Pitts, A.M.: Syntax and Semantics. Part of the documentation for the HOL4 system
8. Pitts, A.M.: Nominal Logic, A First Order Theory of Names and Binding. Information and Computation 183, 165–193 (2003)
9. Sato, M., Pollack, R.: External and Internal Syntax of the Lambda-Calculus. To appear in Journal of Symbolic Computation
10. Tobin-Hochstadt, S., Felleisen, M.: The Design and Implementation of Typed Scheme. In: Proc. of the 35rd Symposium on Principles of Programming Languages (POPL), pp. 395–406. ACM, New York (2008)
11. Urban, C., Cheney, J., Berghofer, S.: Mechanizing the Metatheory of LF. In: Proc. of the 23rd IEEE Symposium on Logic in Computer Science (LICS), pp. 45–56 (2008)
12. Urban, C., Pitts, A., Gabbay, M.: Nominal Unification. Theoretical Computer Science 323 (1-3), 473–497 (2004)
13. Urban, C., Zhu, B.: Revisiting Cut-Elimination: One Difficult Proof is Really a Proof. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 409–424. Springer, Heidelberg (2008)

(Nominal) Unification by Recursive Descent with Triangular Substitutions

Ramana Kumar[1] and Michael Norrish[1,2]

[1] The Australian National University
u4305025@anu.edu.au
[2] Canberra Research Lab., NICTA
Michael.Norrish@nicta.com.au

Abstract. Using HOL4, we mechanise termination and correctness for two unification algorithms, written in a recursive descent style. One computes unifiers for first order terms, the other for nominal terms (terms including α-equivalent binding structure). Both algorithms work with triangular substitutions in accumulator-passing style: taking a substitution as input, and returning an extension of that substitution on success.

1 Introduction

The fastest known first-order unification algorithms are time and space linear (or almost linear) in the size of the input terms [1,2]. In the case of nominal unification, polynomial [3], including quadratic [4], algorithms exist. By comparison, the algorithms in this paper are naïve in two ways: they perform recursive descent of the terms being unified, applying new bindings along the way; and they perform the occurs check with every new binding. Recursive descent interleaved with application can require time exponential in the size of the original terms. Also, it is possible to do the occurs check only once, or even implicitly, in an algorithm that doesn't recursively descend terms.

However, naïve algorithms are used in real systems for a number of reasons: worst case inputs do not arise often in practice, encoding the input and decoding the output of a fast algorithm can be costly, and naïve algorithms are simpler to implement and teach. Some evidence for the first two assertions can be found in Hoder and Voronkov [5] where an imperative version of the algorithm in this paper (there labelled "Robinson's") benchmarks better than the worst-case linear algorithms.[1]

One important feature of the algorithms considered by Hoder and Voronkov is that they all use *triangular* substitutions. This representation is useful in systems that do backtracking and need to "unapply" substitutions from terms, because it enables the use of persistent data structures. The unapply operation becomes implicit (and thus, efficient) when updates are made to a persistent

[1] These benchmarks were made in the context of automated theorem provers with term indexing; we don't consider the maintenance of a term index in this paper.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 51–66, 2010.

substitution: backtracking computations simply apply the appropriate subset of the shared substitution whenever terms in context are required.

A triangular substitution [6] is a set of singleton maps (each binding a different variable). When this set is implemented as a list, update is constant time and sharing is maximised. When using triangular substitutions, and writing in a functional language, it is natural to write unification in an accumulator-passing style. (The analogue in an imperative setting is to update a global variable, which is what happens in the implementation of Robinson's algorithm in Hoder and Voronkov.) So, for example, the unification algorithm in miniKanren [7,8] takes two terms, t_1 and t_2, and an accumulator substitution, s. It returns an extension of s with any new bindings necessary to make t_1 and t_2 unify (or fails if that's impossible).

Triangular substitutions are generally not *idempotent*. For example, a binding from y to z may be added to a substitution already binding x to y. Applying the extended substitution once to x yields y, but applying it twice yields z. But a triangular substitution can represent the same information as an idempotent substitution using exponentially less space. For example if x is bound to the pair (y, y) and y is bound to the ground term $(1, 2)$ then an idempotent substitution would contain three copies of $(1, 2)$, whereas a triangular substitution would contain just one.

Baader and Snyder [6] mention using triangular substitutions in a recursive descent algorithm as a good idea, but do not pursue it because of the exponential time complexity. Our own experiments agree that using triangular substitutions gives better speed and memory usage than computing idempotent substitutions.

Nominal Unification. Classical unification works over first-order terms. Recently, there has been interest in the theory and implementation of logical systems using *nominal* terms, which include names and binders. Such terms provide natural representations of syntaxes occurring in logic and computer science.

Nominal unification was first defined by Urban, Pitts, and Gabbay [9]. Nominal systems (*e.g.*, αProlog [10], alphaKanren [11]) need to be able to unify nominal terms. The mechanisations in this paper are of algorithms inspired by the implementations in miniKanren (first-order) and alphaKanren (nominal). (The alphaKanren paper [11] describes unification with idempotent substitutions; our mechanisation is of a later, more efficient implementation using triangular substitutions.)

Contributions

- We provide mechanised definitions of accumulator-passing style first-order (`unify`) and nominal (`nomunify`) unification algorithms. Both definitions require the provision of a novel termination argument (Section 4).
- Since the unification algorithms may diverge if the accumulator contains loops, we define and characterise a well-formedness condition on triangular substitutions that forbids loops (Section 2), and show that the algorithms preserve it (Sections 5 and 6).

– We mechanise algorithms for applying triangular substitutions, providing the requisite termination arguments in Section 3.
– We provide statements of correctness (soundness, completeness, and generality) for unification algorithms written in accumulator-passing style, and prove them of `unify` in Section 5, and of `nomunify` in Section 6.

The mechanised theories containing all the results in this paper are available online at `https://bitbucket.org/michaeln/formal_mk/src/tip/hol/`. Since the results have been machine-checked, we will omit the proofs of some lemmas.

Notation. In general, higher order logic syntax for Boolean terms uses standard connectives and quantifiers ($\wedge$, $\forall$ *etc.*). Iterated application of a function is written $f^n\ x$, meaning $f(f(\ldots f(x)))$. R^+ denotes the transitive closure of a relation. The relation `measure` f, where f is of type $\alpha \rightarrow$ `num`, relates x and y if $f(x) < f(y)$.

The `do` notation is used for writing in monadic style. We only use it to express bind in the option monad: the term `do` y `<-` $f\ x$`;` $g\ y$ `od` means `NONE` if $f\ x$ returns `NONE`. Otherwise, if $f\ x$ returns `SOME` y, then the term is the result of applying g to y (giving a value of option type, either `NONE` or `SOME` v).

`FLOOKUP` $fm\ k$ applies a finite map, returning `SOME` v when the key is in the domain, otherwise `NONE`. The domain of a finite map is written `FDOM` fm. The sub-map relation is written $fm_1 \sqsubseteq fm_2$. The empty finite map is written `FEMPTY`. The update of a finite map with a new key-value pair is written fm `|+ (`k`,`v`)`. Composition of a function after a finite map is written $f \circ fm$.

Tuples and inductive data types can be deconstructed by case analysis. The notation is `case` t_1 `of` $p_1 \rightarrow e_1 \parallel p_2 \rightarrow e_2$. Patterns may include underscores as wildcards.

For each type, the constant `ARB` denotes an arbitrary object of that type.

2 Terms and Substitutions

The word "substitution" can refer to an action—substitution of t_1 for x and t_2 for y in t—or it can refer to an object, a collection of variable bindings—the substitution that binds x to t_1 and y to t_2. When viewing substitutions as data structures containing bindings, a separate function is used to apply a substitution to a term to produce a new term. Equivalent substitutions, under application, may be different as data structures. We distinguish substitutions from substitution application in order to investigate a representation, triangular form, suited to the functional programming idiom of implicitly shared data.

We define first-order terms inductively as follows. We represent variables by natural numbers (strings would be equally good). Terms are parameterised by the type α representing constant values (*e.g.*, function symbols).

Definition 1. *Terms*

term `= Var of num | Pair of` α `term =>` α `term | Const of` α

We represent a substitution (HOL type α `subst`) as a finite map from numbers to terms, thereby abstracting over any particular data structure (an association list or something more sophisticated) without losing the distinction between a substitution and its application. Application to a term is defined as follows. We will define a different notion of substitution application more suited to triangular substitutions in Section 3.

Definition 2. *Substitution application*

$$s \texttt{ ' (Var } v\texttt{)} = \texttt{case FLOOKUP } s\ v \texttt{ of NONE} \rightarrow \texttt{Var } v \parallel \texttt{SOME } t \rightarrow t$$
$$s \texttt{ ' (Pair } t_1\ t_2\texttt{)} = \texttt{Pair } (s \texttt{ ' } t_1)\ (s \texttt{ ' } t_2)$$
$$s \texttt{ ' (Const } c\texttt{)} = \texttt{Const } c$$

A substitution is `idempotent` if repeated application is the same as a single application. Applying a substitution to a variable outside its domain yields that variable. But our representation permits a substitution explicitly binding a variable to itself. We will exclude such substitutions with the condition `noids` s.

The application of a substitution s to itself is obtained by replacing every term t in the range of s by its image under s. This is the closest we will get to substitution composition (`selfapp` s is s composed with itself); instead we will compose application functions.

Lemma 1. $\vdash$ `selfapp` s `'` $t = s$ `'` $(s$ `'` $t)$

Well-formed Substitutions

For each substitution s we define a relation $\texttt{tri}_\texttt{R}\ s$ that holds between a variable in the domain and a variable in the corresponding term.

Definition 3. *Relating a variable to those in the term to which it's bound*

$$\texttt{tri}_\texttt{R}\ s\ y\ x \iff$$
$$\texttt{case FLOOKUP } s\ x \texttt{ of NONE} \rightarrow \texttt{F} \parallel \texttt{SOME } t \rightarrow y \in \texttt{vars } t$$

A substitution is well-formed (`wfs`) if $\texttt{tri}_\texttt{R}\ s$ is well-founded. There are three informative statements equivalent to the well-formedness of a substitution.

Lemma 2. *Only well-formed substitutions have no cycles*

$$\vdash \texttt{wfs } s \iff \forall v.\ \neg(\texttt{tri}_\texttt{R}\ s)^+\ v\ v$$

Corollary 1. $\vdash$ `wfs` $s \Rightarrow$ `noids` s

Lemma 3. *Only well-formed substitutions are well-formed after self-application*

$$\vdash \texttt{wfs } s \iff \texttt{wfs (selfapp } s\texttt{)}$$

Lemma 4. *Only well-formed substitutions have fixpoints*

$$\vdash \texttt{wfs } s \iff \exists n.\ \texttt{idempotent (selfapp}^n\ s\texttt{)} \wedge \texttt{noids (selfapp}^n\ s\texttt{)}$$

Proof. From right to left, the result follows by induction on n. From left to right, the `noids` condition follows from Lemma 3 and Corollary 1. Idempotence follows by contradiction. If a substitution is not idempotent there will be a variable that maps to a term including a variable in the substitution's domain. If this occurs within a substitution iterated n times, there must be a chain of length n within the original substitution with the same property. But an arbitrarily long chain cannot exist without a loop, contradicting our well-formedness assumption.

Lemma 4 (with Lemma 1) shows that well-formedness is necessary and sufficient for being able to recover an equivalent idempotent substitution.

3 Substitution Application

Since we are interested in maintaining triangular substitutions, we want to be able to apply a non-idempotent substitution as if we had collapsed it down to an idempotent one by repeated self-application without actually doing so. This is achieved by recursion in the application function `walk*` (we write $s \triangleleft t$ for the application of `walk*` to substitution s and term t): if we encounter a variable in the domain of the substitution, we look it up and recur on the result. Defining this function presents the first of a number of interesting termination problems.

The clearest expression of `walk*`'s behaviour is the following characterisation:

Lemma 5. *Characterisation of* `walk*`

```
⊢ wfs s ⇒
  s ◁ Var v =
    (case FLOOKUP s v of NONE → Var v ‖ SOME t → s ◁ t) ∧
  s ◁ Pair t1 t2 = Pair (s ◁ t1) (s ◁ t2) ∧
  s ◁ Const c = Const c
```

Lemma 6. `walk*` *reduces to application on idempotent substitutions*

```
⊢ wfs s ⇒ (idempotent s ⟺ walk* s = (') s)
```

The `walk*` function can be viewed as performing a tree traversal ("walk") of its eventual output term. Other algorithms, including `unify`, need to perform some of this tree walk, but may not need to immediately traverse a term to its leaves. We isolate the part of `walk*` that finds the ultimate binding of a variable, calling this `vwalk`:

Definition 4. *Walking a variable*

```
wfs s ⇒
vwalk s v =
  case FLOOKUP s v of
    SOME (Var u) → vwalk s u
  ‖ SOME t → t
  ‖ NONE → Var v
```

Proving termination for `vwalk` under the assumption `wfs` s follows easily from the definitions.

Following the miniKanren code, we define a function `walk`, which either calls `vwalk` if its argument is a variable, or returns its argument. It is a common miniKanren idiom (used in `unify`, among other places) to begin functions by walking term arguments in the current substitution. This reveals just enough of a term-in-context's structure for the current level of recursion. This idiom is used in the definition of `walk*`, which can be stated thus:

Definition 5. *Substitution application, walking version*

```
wfs s ⇒
s ◁ t =
  case walk s t of
    Pair t1 t2 → Pair (s ◁ t1) (s ◁ t2)
  ‖ t' → t'
```

The termination relation for `walk*` is the lexicographic combination of the multiset ordering with respect to $(\texttt{tri}_R\ s)^+$ over a term's variables, and the term's size.

The "walk first" idiom is also used to define the occurs-check. We omit the definition but provide the following characterisation.

Lemma 7. *The occurs-check finds variables in the term after application*

$\vdash$ `wfs` $s \Rightarrow$ `(oc` $s\ t\ v \iff v \in$ `vars (`$s \triangleleft t$`))`

4 Unification: Definition

Our unification algorithm, `unify`, has type

α `subst` $\rightarrow$ α `term` $\rightarrow$ α `term` $\rightarrow$ α `subst option`

The option type in the result is used to signal whether or not the input terms are unifiable. We accept that `unify` will have an undefined value when given a malformed substitution as input. Our strategy for defining `unify` is to define a total version, `tunify`; to extract and prove the termination conditions; and to then show that `unify` exists and equals `tunify` for well-formed substitutions. The definition of `tunify` is given in Figure 1.

Three termination conditions are generated by HOL4, corresponding to the need for a well-founded relation and the two recursive calls:

1. `WF` R
2. $\forall t_2\ t_1\ s\ t_{11}\ t_{12}\ t_{21}\ t_{22}.$
   `wfs` $s \wedge$ `walk` $s\ t_1 =$ `Pair` $t_{11}\ t_{12} \wedge$ `walk` $s\ t_2 =$ `Pair` $t_{21}\ t_{22} \Rightarrow$
   R `(`s`,`t_{11}`,`t_{21}`) (`s`,`t_1`,`t_2`)`
3. $\forall t_2\ t_1\ s\ t_{11}\ t_{12}\ t_{21}\ t_{22}\ sx.$
   `wfs` $s \wedge$ `(walk` $s\ t_1 =$ `Pair` $t_{11}\ t_{12} \wedge$ `walk` $s\ t_2 =$ `Pair` $t_{21}\ t_{22}$`) ∧`
   `tunify_tupled_aux` R `(`s`,`t_{11}`,`t_{21}`) = SOME` $sx \Rightarrow$
   R `(`sx`,`t_{12}`,`t_{22}`) (`s`,`t_1`,`t_2`)`

Definition 6. *Unification with triangular substitutions (total version)*

```
tunify s t1 t2 =
  if wfs s then
    case (walk s t1,walk s t2) of
      (Var v1,Var v2) →
        SOME (if v1 = v2 then s else s |+ (v1,Var v2))
    ‖ (Var v1,t2) →
        if oc s t2 v1 then NONE else SOME (s |+ (v1,t2))
    ‖ (t1,Var v2) →
        if oc s t1 v2 then NONE else SOME (s |+ (v2,t1))
    ‖ (Pair t11 t12,Pair t21 t22) →
        do sx <- tunify s t11 t21; tunify sx t12 t22 od
    ‖ (Const c1,Const c2) → if c1 = c2 then SOME s else NONE
    ‖ _ → NONE
  else
    ARB
```

Fig. 1. First-Order Unification: the `unify` function is the `then` branch of the `if`

A call `tunify_tupled_aux` R *args* is a guarded call to the only-partially defined `tunify`: any recursive calls must be on arguments that are R-smaller than *args*. The call appears in Condition 3 because the argument sx in the second recursive call `tunify` sx t_{12} t_{22} is the result of the first recursive call. This is thus an instance of nested recursion.

The `unify` function walks the subterms being considered in the current substitution before case analysis. The key to the termination argument is that size of the subterms, considered in the context of the updated substitution, goes down on every recursive call. The termination relation $\texttt{unify}_\texttt{R}$, defined below, makes this statement in the final conjunct. The other conjuncts are also satisfied by the algorithm and are required to ensure that $\texttt{unify}_\texttt{R}$ is well-founded.

Definition 7. *Termination relation for* `unify`

$$\texttt{unify}_\texttt{R}\ (sx,c_1,c_2)\ (s,t_1,t_2) \iff \texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge \texttt{allvars}\ sx\ c_1\ c_2 \subseteq \texttt{allvars}\ s\ t_1\ t_2 \wedge \texttt{measure}\ (\texttt{term\_depth} \circ \texttt{walk}^*\ sx)\ c_1\ t_1$$

Theorem 1. $\texttt{unify}_\texttt{R}$ *is well-founded*

$$\vdash \texttt{WF}\ \texttt{unify}_\texttt{R}$$

Proof. By contradiction. If there is an infinite $\texttt{unify}_\texttt{R}$-chain, then the set of variables in the arguments (`allvars`) must reach a fixpoint because each successive set is a subset of its predecessor, and the sets are finite. As the set of variables is getting smaller, the substitutions are allowed to get larger (the $\sqsubseteq$ relation). However, once the set of variables reaches its fixpoint, the substitutions will be drawing on a fixed source for new variable bindings, so they must also reach a fixpoint. Once the substitution (sx) is fixed, the first argument of the `measure`

conjunct becomes fixed. Hence the supposedly infinite chain would have to stop (when $sx \lhd c_1$ has zero depth): contradiction.

We thereby satisfy Termination Condition 1. Condition 2 is easy because the substitution doesn't change.

Lemma 8. *Termination Condition 2*

$$\vdash \texttt{wfs}\ s \wedge \texttt{walk}\ s\ t_1 = \texttt{Pair}\ t_{11}\ t_{12} \wedge \texttt{walk}\ s\ t_2 = \texttt{Pair}\ t_{21}\ t_{22} \Rightarrow \texttt{unify}_\texttt{R}\ (s, t_{11}, t_{21})\ (s, t_1, t_2)$$

Proof. For the conjunct involving `allvars`: either $t_1 = \texttt{Pair}\ t_{11}\ t_{12}$ or the pair is in the range of the substitution, and similarly for t_2. The other $\texttt{unify}_\texttt{R}$ conjuncts are simple.

Condition 3, however, requires some work. We define another relation, $\texttt{subst}_\texttt{R}$, weaker than $\texttt{unify}_\texttt{R}$, which asserts that the variables of the result substitution all come from the arguments. The $\texttt{subst}_\texttt{R}$ relation serves as a bridge: weak enough that we can prove it is satisfied by `tunify` by induction and strong enough that it implies $\texttt{unify}_\texttt{R}$. We use a relation that restricts the substitution only since at this point we can't say much about recursive calls without proving $\texttt{unify}_\texttt{R}$ for each call.

Definition 8. *Relation between the output substitution and input arguments*

$$\texttt{subst}_\texttt{R}\ sx\ s\ t_1\ t_2 \iff \texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge \texttt{substvars}\ sx \subseteq \texttt{allvars}\ s\ t_1\ t_2$$

Lemma 9. $\texttt{subst}_\texttt{R}$ *implies* $\texttt{unify}_\texttt{R}$ *on subterms*

$$\vdash \texttt{wfs}\ s \wedge \texttt{walk}\ s\ t_1 = \texttt{Pair}\ t_{11}\ t_{12} \wedge \texttt{walk}\ s\ t_2 = \texttt{Pair}\ t_{21}\ t_{22} \wedge (\texttt{subst}_\texttt{R}\ sx\ s\ t_{11}\ t_{21} \vee \texttt{subst}_\texttt{R}\ sx\ s\ t_{12}\ t_{22}) \Rightarrow \texttt{unify}_\texttt{R}\ (sx, t_{12}, t_{22})\ (s, t_1, t_2)$$

Lemma 10. `unify` *implies* $\texttt{subst}_\texttt{R}$

$$\vdash \texttt{wfs}\ s \wedge \texttt{tunify\_tupled\_aux}\ \texttt{unify}_\texttt{R}\ (s, t_1, t_2) = \texttt{SOME}\ sx \Rightarrow \texttt{subst}_\texttt{R}\ sx\ s\ t_1\ t_2$$

Proof. By well-founded induction (knowing that $\texttt{unify}_\texttt{R}$ is well-founded).

Lemma 11. *Termination Condition 3*

$$\vdash \texttt{wfs}\ s \wedge \texttt{walk}\ s\ t_1 = \texttt{Pair}\ t_{11}\ t_{12} \wedge \texttt{walk}\ s\ t_2 = \texttt{Pair}\ t_{21}\ t_{22} \wedge \texttt{tunify\_tupled\_aux}\ \texttt{unify}_\texttt{R}\ (s, t_{11}, t_{21}) = \texttt{SOME}\ sx \Rightarrow \texttt{unify}_\texttt{R}\ (sx, t_{12}, t_{22})\ (s, t_1, t_2)$$

Proof. From the lemmas above.

5 Unification: Correctness

There are three parts to the correctness statement: if `unify` succeeds then its result is a unifier; if `unify` succeeds then its result is most general; and if there exists a unifier of $s \lhd t_1$ and $s \lhd t_2$, then `unify` $s\ t_1\ t_2$ succeeds. A substitution s is a *unifier* of terms t_1 and t_2 if $s \lhd t_1 = s \lhd t_2$.

It is not generally true that the result of `unify` is idempotent. But `unify` preserves well-formedness, which (as per Lemma 4) ensures the well-formed result can be collapsed into an idempotent substitution.

Theorem 2. *The result of* `unify` *is a unifier and a well-formed extension*

$$\vdash \texttt{wfs}\ s \wedge \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx \Rightarrow \\ \texttt{wfs}\ sx \wedge s \sqsubseteq sx \wedge sx \lhd t_1 = sx \lhd t_2$$

Proof. The first two conjuncts, that s is a sub-map of sx and sx is well-formed, are corollaries of Lemma 10. Essentially, `unify` only updates the substitution, and then only with variables that aren't already in the domain.

The rest follows by recursion induction on `unify`, using Lemma 12 (below), which states that applying a sub-map of a substitution, and then the larger substitution, is the same as simply applying the larger substitution on its own.

Lemma 12. `walk*` *over a sub-map*

$$\vdash s \sqsubseteq sx \wedge \texttt{wfs}\ sx \Rightarrow sx \lhd t = sx \lhd (s \lhd t)$$

Corollary 2. `walk*` *with a fixed substitution is idempotent*

Given Lemma 12 and Theorem 2, we can equally regard `unify` $s\ t_1\ t_2$ as calculating a unifier for t_1 and t_2 or for the terms-in-context $s \lhd t_1$ and $s \lhd t_2$.

The context provided by the input substitution is relevant to our notion of a most general unifier, which differs from the usual context-free notion. A unifier of terms in context is *most general* if it can be composed with another substitution to equal any other unifier *in the same context*. In the empty context, however, the notions of most general unifier coincide.

Lemma 13. *The kinds of extensions made by unify are innocuous*

$$\vdash \texttt{wfs}\ s_1 \wedge \texttt{wfs}\ (s\ \texttt{|+}\ (vx, tx)) \wedge vx \notin \texttt{FDOM}\ s \wedge \\ s_1 \lhd \texttt{Var}\ vx = s_1 \lhd (s \lhd tx) \Rightarrow \\ \forall t.\ s_1 \lhd (s\ \texttt{|+}\ (vx, tx) \lhd t) = s_1 \lhd (s \lhd t)$$

Proof. By recursion induction on `walk*`.

Lemma 14. *The result of* `unify` *is most general (in context)*

$$\vdash \texttt{wfs}\ s \wedge \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx \wedge \texttt{wfs}\ s_2 \wedge \\ s_2 \lhd (s \lhd t_1) = s_2 \lhd (s \lhd t_2) \Rightarrow \\ \forall t.\ s_2 \lhd (sx \lhd t) = s_2 \lhd (s \lhd t)$$

Proof. By recursion induction on `unify` using the lemma above.

Theorem 3. *The result of* `unify` *is most general (empty context)*

$$\vdash \texttt{unify FEMPTY}\ t_1\ t_2 = \texttt{SOME}\ sx \Rightarrow \\ \forall s.\ \texttt{wfs}\ s \wedge s \lhd t_1 = s \lhd t_2 \Rightarrow \exists s'.\ \forall t.\ s' \lhd (sx \lhd t) = s \lhd t$$

Remark 1. By the lemma above we see that the witness is s itself.

We now turn to the third correctness result.

Lemma 15. *A variable and a term containing that variable remain different under application*

$$\vdash \texttt{oc}\ s\ t\ v \wedge (\forall w.\ t \neq \texttt{Var}\ w) \wedge \texttt{wfs}\ s \wedge \texttt{wfs}\ s_2 \Rightarrow \\ s_2 \lhd \texttt{Var}\ v \neq s_2 \lhd (s \lhd t)$$

Proof. By considering the term sizes.

Theorem 4. *If the terms are unifiable, then* `unify` *succeeds*

$$\vdash \texttt{wfs}\ s \wedge \texttt{wfs}\ s_2 \wedge s_2 \lhd (s \lhd t_1) = s_2 \lhd (s \lhd t_2) \Rightarrow \\ \exists sx.\ \texttt{unify}\ s\ t_1\ t_2 = \texttt{SOME}\ sx$$

Proof. By recursion induction on `unify`, using Lemma 15 for the non-trivial occurs checks, and using Lemma 14 for the recursive case.

6 Nominal Unification

Nominal terms extend first-order terms with two new constructors, one for names (also called atoms), and one for *ties*, which represent binders (terms with a bound name). We also replace the `Var` constructor with a constructor for *suspensions*, the nominal analogue of variables. A suspension is made up of a variable name and a permutation of names, and stands for the variable after application of the permutation. When (if) the variable is bound, the permutation can be applied further.

Definition 9. *Concrete nominal terms*

Cterm

```
= CNom of string
| CSus of (string, string) alist => num
| CTie of string => α Cterm
| CPairn of α Cterm => α Cterm
| CConstn of α
```

We represent permutations as lists of pairs of names; such a list stands for a ordered composition of swaps, with the head of list applied last. There may be more than one list representing the same permutation. We abstract over these different lists by creating a quotient type. The nominal term data type is the quotient of the concrete type above by permutation equivalence `(==)`.

Constructors in the quotient type are the same as in the concrete type but with the `C` prefix removed.

Following the example of the first-order algorithm, we begin by defining the "walk" operation that finds a suspension's ultimate binding:

Definition 10. *Walking a suspension*

```
wfs_n s ⇒
vwalk_n s π v =
  case FLOOKUP s v of
    SOME (Sus p u) → vwalk_n s (π ++ p) u
  | SOME t → π • t
  | NONE → Sus π v
```

The π `++` p term appends π and p, producing their composition; $\pi \bullet t$ is the (homomorphic) application of a permutation to a term.

The termination argument for $\texttt{vwalk}_\texttt{n}$ is the same as in the first-order case; the permutation doesn't play a part in the recursion. Substitution application is analogous to the first-order case: $\texttt{walk}_\texttt{n}$ calls $\texttt{vwalk}_\texttt{n}\ s\ p\ v$ for a suspension `Sus` $p\ v$, otherwise returns its argument; $\texttt{walk}^*_\texttt{n}$ uses $\texttt{walk}_\texttt{n}$, recurring on ties as well as pairs.

In the first phase of nominal unification (as defined in [9]), a substitution is constructed along with a set of *freshness constraints* (alternatively, a *freshness environment*). A freshness constraint is a pair of a name and a variable, expressing the constraint that the variable is never bound to a term in which the name is free.

The second phase of unification checks to see if the freshness constraints are consistent, possibly dropping irrelevant constraints along the way. If this check succeeds, the substitution and the new freshness environment, which together form a nominal unifier, are returned. The use of triangular substitutions and the accumulator-passing style means that our definition of nominal unification differs from [9] in that the substitution returned from the first phase must be referred to as the freshness constraints are checked in the second phase.

The final definition in HOL is presented in Definition 12 (Figure 2). In both phases, we use the auxiliary `term_fcs`. This function is given a name and a term, and constructs a minimal freshness environment sufficient to ensure that the name is fresh for the term. If this is impossible (*i.e.*, if the name is free in the term), `term_fcs` returns `NONE`.

Following our strategy in the first-order case, $\texttt{unify}_\texttt{n}$ is defined *via* a total function $\texttt{tunify}_\texttt{n}$. The pair and constant cases are unchanged, and names are treated as constants. With suspensions, there is an extra case to consider: if the variables are the same, we augment the freshness environment with a constraint $(a, s \lhd_\texttt{n} \texttt{Sus []}\ v)$ for every name a in the disagreement set of the permutations (done by `unify_eq_vars`). In the other suspension cases, we apply the inverse (reverse) of the suspension's permutation to the term before performing the binding (done in `add_bdg`). (We invert the permutation so that applying the permutation to the term to which the variable is bound results in the term with which the suspension is supposed to unify.)

In the `Tie` case, a simple recursive descent is possible when the bound names are the same. Otherwise, we ensure that the first name is fresh for the body of the second term, and swap the two names in the second term before recursing.

Phase 2 is implemented by `verify_fcs`, which calls

```
term_fcs a (s ◁n Sus [] v)
```

for each constraint (a, v) in the environment, accumulating the result.

Termination The termination argument for Phase 1 is analogous to the termination argument for `unify` in the first-order case. We use the same termination relation (this time measuring nominal term depth, and ignoring the freshness environment). The extra termination condition for recursion down a `Tie` is handled like the easier of the `Pair` conditions because the substitution doesn't change and the freshness environment is irrelevant to termination.

Termination for Phase 2 depends only on the freshness environment being finite. We assume the freshness environment is finite in all valid inputs to `nomunify`, and it's easy to show that `term_fcs` (and hence Phase 1) preserves finiteness by structural induction on the nominal term.

6.1 Correctness

In the first-order case, unified terms are syntactically equal. In the nominal case, unified terms must be α-equivalent with respect to a freshness environment, written $fe \vdash t_1 \approx t_2$. For example, $(\lambda a.X)$ and $(\lambda b.Y)$ unify with X bound to $(a\,b)\cdot Y$ (the substitution), but only if $a\#Y$ (a singleton freshness environment). In the absence of the latter, one might instantiate Y with a, and therefore X with b, producing non-equivalent terms. We write $fe \vdash a \# t$ to mean that a name is fresh for a term with respect to a freshness environment.

Lemma 16. *The freshness environment computed by* `unify_eq_vars` *makes the suspensions equivalent*

```
⊢ wfsn s ∧
  unify_eq_vars (dis_set π1 π2) v (s,fe) = SOME (s,fcs) ⇒
  fcs ⊢ s ◁n Sus π1 v ≈ s ◁n Sus π2 v
```

Lemma 17. `verify_fcs` *extends equivalence to terms under the substitution*

```
⊢ fe ⊢ t1 ≈ t2 ∧ wfsn s ∧ FINITE fe ∧
  verify_fcs fe s = SOME fex ⇒
  fex ⊢ s ◁n t1 ≈ s ◁n t2
```

Lemma 18. *The result of* `verify_fcs` *in a sub-map can be verified in the extension*

```
⊢ verify_fcs fe s = SOME ve0 ∧ verify_fcs fe sx = SOME ve ∧
  s ⊑ sx ∧ wfsn sx ∧ FINITE fe ⇒
  verify_fcs ve0 sx = SOME ve
```

Corollary 3. `verify_fcs` *with a fixed substitution is idempotent.*

Definition 11. *Phase 1 (total version)*

```
add_bdg π v t0 (s,fe) =
  (let t = π^-1 • t0 in
     if oc_n s t v then NONE else SOME (s |+ (v,t),fe))

tunify_n (s,fe) t1 t2 =
  if wfs_n s then
    case (walk_n s t1,walk_n s t2) of
      (Nom a1,Nom a2) → if a1 = a2 then SOME (s,fe) else NONE
    ‖ (Sus π1 v1,Sus π2 v2) →
        if v1 = v2 then
          unify_eq_vars (dis_set π1 π2) v1 (s,fe)
        else
          add_bdg π1 v1 (Sus π2 v2) (s,fe)
    ‖ (Sus π1 v1,t2) → add_bdg π1 v1 t2 (s,fe)
    ‖ (t1,Sus π2 v2) → add_bdg π2 v2 t1 (s,fe)
    ‖ (Tie a1 t1,Tie a2 t2) →
        if a1 = a2 then
          tunify_n (s,fe) t1 t2
        else
          do
            fcs <- term_fcs a1 (s ◁_n t2);
            tunify_n (s,fe ∪ fcs) t1 ([(a1,a2)] • t2)
          od
    ‖ (Pair_n t11 t12,Pair_n t21 t22) →
        do
          (sx,fex) <- tunify_n (s,fe) t11 t21;
          tunify_n (sx,fex) t12 t22
        od
    ‖ (Const_n c1,Const_n c2) →
        if c1 = c2 then SOME (s,fe) else NONE
    ‖ _ → NONE
  else
    ARB
```

Definition 12. *Nominal unification in two phases*

```
nomunify (s,fe) t1 t2 =
  do
    (sx,feu) <- unify_n (s,fe) t1 t2;
    fex <- verify_fcs feu sx;
    SOME (sx,fex)
  od
```

Fig. 2. Nominal Unification

Theorem 5. *The result of* `nomunify` *is a unifier, the freshness environment is finite, and the substitution is a well-formed extension*

$$\begin{array}{l}\vdash \texttt{wfs}_\texttt{n}\ s \wedge \texttt{FINITE}\ fe \wedge \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \Rightarrow \\ \quad \texttt{FINITE}\ fex \wedge \texttt{wfs}_\texttt{n}\ sx \wedge s \sqsubseteq sx \wedge fex \vdash sx \lhd_\texttt{n} t_1 \approx sx \lhd_\texttt{n} t_2\end{array}$$

Proof. By recursion induction on `unify` *using the lemmas above.*

Theorem 6. *The result of* `nomunify` *is most general*

$$\begin{array}{l}\vdash \texttt{wfs}_\texttt{n}\ s \wedge \texttt{FINITE}\ fe \wedge \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex) \wedge \\ \quad \texttt{wfs}_\texttt{n}\ s_2 \wedge fe_2 \vdash s_2 \lhd_\texttt{n} (s \lhd_\texttt{n} t_1) \approx s_2 \lhd_\texttt{n} (s \lhd_\texttt{n} t_2) \Rightarrow \\ \quad (\forall a\ v. \\ \qquad (a,v) \in fex \Rightarrow \\ \qquad (\exists b\ w\ fcs. \\ \qquad\quad (b,w) \in fe \wedge \texttt{term\_fcs}\ b\ (sx \lhd_\texttt{n} \texttt{Sus}\ []\ w) = \texttt{SOME}\ fcs \wedge \\ \qquad\quad (a,v) \in fcs) \vee fe_2 \vdash a \# s_2 \lhd_\texttt{n} \texttt{Sus}\ []\ v) \wedge \\ \quad \forall t.\ fe_2 \vdash s_2 \lhd_\texttt{n} (sx \lhd_\texttt{n} t) \approx s_2 \lhd_\texttt{n} (s \lhd_\texttt{n} t)\end{array}$$

Proof. The second part of the conclusion is analogous to Lemma 14, and the proof is similar.

The first part is *via* the following lemma, which is proved by recursion induction on `unify`: that any freshness constraint generated by `nomunify` either originates in the input environment or is a member of the minimal freshness environment required to equate $sx \lhd_\texttt{n} t_1$ and $sx \lhd_\texttt{n} t_2$. We then use the second part to show that (s_2,fe_2) must satisfy that minimal environment.

Theorem 7. *If the terms are unifiable, then* `nomunify` *succeeds*

$$\begin{array}{l}\vdash fe_2 \vdash s_2 \lhd_\texttt{n} (s \lhd_\texttt{n} t_1) \approx s_2 \lhd_\texttt{n} (s \lhd_\texttt{n} t_2) \wedge \texttt{wfs}_\texttt{n}\ s_2 \wedge \texttt{wfs}_\texttt{n}\ s \Rightarrow \\ \quad \exists sx. \\ \qquad \forall fe. \\ \qquad\quad \texttt{FINITE}\ fe \wedge \texttt{verify\_fcs}\ fe\ sx \neq \texttt{NONE} \Rightarrow \\ \qquad\quad \exists fex.\ \texttt{nomunify}\ (s,fe)\ t_1\ t_2 = \texttt{SOME}\ (sx,fex)\end{array}$$

We can always provide the empty set for the input freshness environment, since $\texttt{verify\_fcs}\ \emptyset\ s = \texttt{SOME}\ \emptyset$.

Proof. By recursion induction on `unify`; the proof is similar to that of Theorem 4. Since `unify` extends but otherwise ignores the input freshness environment, we assume the input freshness environment is empty for the inductive proof. We also use the following lemma in the recursive case.

Lemma 19. *The freshness environment generated by one side of a pair will verify in the substitution computed for both sides.*

$$\begin{array}{l}\vdash fe \vdash t_1 \approx t_2 \wedge \texttt{term\_fcs}\ a\ t_1 = \texttt{SOME}\ fcs_1 \wedge fcs_1 \subseteq fe \Rightarrow \\ \quad \exists fcs_2.\ \texttt{term\_fcs}\ a\ t_2 = \texttt{SOME}\ fcs_2 \wedge fcs_2 \subseteq fe\end{array}$$

7 Related Work

Robinson's recursive descent algorithm traditionally takes two terms as input and produces an idempotent most general unifier on success. This algorithm has been mechanised elsewhere in an implementable style (*e.g.*, by Paulson [12]). McBride [13] shows that the algorithm can be structurally recursive in a dependently typed setting, and formalises it this way using LEGO. McBride also points to many other formalisations. The other main approach to the presentation and formalisation of unification algorithms is the Martelli-Montanari transformation system [1]. Ruiz-Reina *et al.* [14] formalise a quadratic unification algorithm (using term graphs, due to Corbin and Bidoit) in ACL2 in the transformation style.

Urban [15] formalised nominal unification in Isabelle/HOL in transformation style. Nominal unification admits first-order unification as a special case, so this can also be seen as a formalisation of first-order unification. Much work on implementing and improving nominal unification has been done by Calvès and Fernández. They implemented nominal unification [16] and later proved that the problem admits a polynomial time solution [3] using graph-rewriting.

8 Conclusion

This paper has demonstrated that the pragmatically important technique of the triangular substitution is amenable to formal proof. Unification algorithms using triangular substitutions occur in the implementations of logical systems, and are thus of central importance. We have shown correctness results for unification algorithms in this style, both for the traditional first-order case, and for nominal terms.

Future Work. There are imperative unification algorithms (such as Paterson and Wegman's [2]) with much better time complexity than Robinson's that use ephemeral data structures. Conchon and Filliâtre [17] have shown that Tarjan's classic union-find algorithm can be transformed into one using persistent data structures. It would be interesting to see if similar ideas can be applied to an imperative unification algorithm; indeed some unification algorithms make use of union-find.

The walk-based substitution application algorithms in this paper can benefit from sophisticated representations of substitutions, as well as from optimizations to the walk algorithm itself. We have done some work on formalising the improvements to `walk*` described by Byrd [8]. Future work includes continuing this formalisation and also investigating representations of triangular substitutions other than the obvious lists.

The Martelli-Montanari transformation system has become a standard platform for presenting unification algorithms, but wasn't immediately applicable for us because it assumes idempotent substitutions are used. However it may be possible to create a transformation system based on triangular substitutions, and it would be interesting to see how it relates to the usual system.

In this paper we formalised the original, inefficient presentation of nominal unification from [9]. The improved nominal unification algorithms by Calvès and Fernández should also be formalised.

Acknowledgements. 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.

References

1. Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Trans. Program. Lang. Syst. 4(2), 258–282 (1982)
2. Paterson, M., Wegman, M.N.: Linear unification. J. Comput. Syst. Sci. 16(2), 158–167 (1978)
3. Calvès, C., Fernández, M.: A polynomial nominal unification algorithm. Theor. Comput. Sci. 403(2-3), 285–306 (2008)
4. Levy, J., Villaret, M.: Nominal unification from a higher-order perspective. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 246–260. Springer, Heidelberg (2008)
5. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: Mertsching, B., Hund, M., Aziz, M.Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 435–443. Springer, Heidelberg (2009)
6. Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 445–532. Elsevier and MIT Press (2001)
7. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. The MIT Press, Cambridge (2005)
8. Byrd, W.E.: Relational Programming in miniKanren: Techniques, Applications, and Implementations. PhD thesis, Indiana University (2009)
9. Urban, C., Pitts, A.M., Gabbay, M.: Nominal unification. Theor. Comput. Sci. 323(1-3), 473–497 (2004)
10. Cheney, J., Urban, C.: alpha-Prolog: A logic programming language with names, binding and α-equivalence. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 269–283. Springer, Heidelberg (2004)
11. Byrd, W.E., Friedman, D.P.: alphaKanren: A fresh name in nominal logic programming languages. Scheme and Functional Programming (2007)
12. Paulson, L.C.: Verifying the unification algorithm in LCF. Sci. Comput. Program. 5(2), 143–169 (1985)
13. McBride, C.: First-order unification by structural recursion. J. Funct. Program. 13(6), 1061–1075 (2003)
14. Ruiz-Reina, J.L., Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M.J.: Formal correctness of a quadratic unification algorithm. J. Autom. Reasoning 37(1-2), 67–92 (2006)
15. Urban, C.: Nominal unification (2004), `http://www4.in.tum.de/~urbanc/Unification/`
16. Calvès, C., Fernández, M.: Implementing nominal unification. Electr. Notes Theor. Comput. Sci. 176(1), 25–37 (2007)
17. Conchon, S., Filliâtre, J.C.: A persistent union-find data structure. In: Russo, C.V., Dreyer, D. (eds.) ML, pp. 37–46. ACM, New York (2007)

A Formal Proof of a Necessary and Sufficient Condition for Deadlock-Free Adaptive Networks*

Freek Verbeek[1,2] and Julien Schmaltz[1,2]

[1] Radboud University Nijmegen
Institute for Computing and Information Sciences
P.O. Box 9010 6500GL Nijmegen, The Netherlands
[2] Open University of the Netherlands
School of Computer Science
P.O. Box 6401DL Heerlen, The Netherlands

Abstract. Deadlocks occur in interconnection networks as messages compete for free channels or empty buffers. Deadlocks are often associated with a circular wait between processes and resources. In the context of networks, Duato proved that for adaptive routing networks a cyclic dependency is not sufficient to create a deadlock. He proposed deadlock-free routing techniques allowing cyclic dependencies between channels or buffers. His work was a breakthrough. It was also counterintuitive and only a complex mathematical proof could convince his peers about the soundness of his theory. We define a necessary and sufficient condition that captures Duato's intuition but that is more intuitive and leads to a simpler proof. However, our condition is logically equivalent to Duato's one. We used the ACL2 theorem proving system to formalize our condition and its proof. In particular, we used two features of ACL2, namely the encapsulation principle and quantifiers, to perform an elegant formalization based on second order functions.

1 Introduction

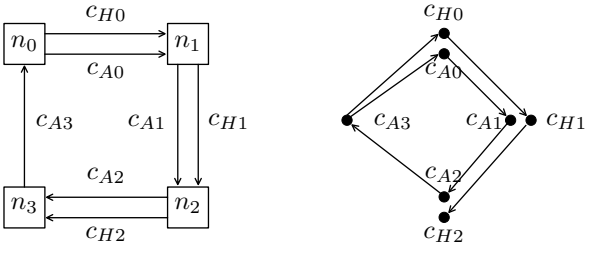
A deadlock is a situation where a set of processes is permanently blocked and no progress is ever possible. This can occur due to a competition for finite resources or reciprocal communications. Classically a deadlock is associated with a circular wait between processes: each process holds a resource needed by the next process [1]. In the context of interconnection networks processes are messages and resources are buffers or channels. A deadlock can occur as messages compete either for free channels or empty buffers. The dependencies between resources are captured by a *channel (or buffer) dependency graph*. Early work by Dally and Seitz has shown that an acyclic dependency graph is a necessary and sufficient condition for deadlock-free routing [2].

This original condition only applies to *deterministic* routing functions, where a message can take only one route from source to destination. An interconnection

* This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 67–82, 2010.

network can have an *adaptive* routing function. If a message is blocked on its way, an adaptive routing function proposes an alternative next-hop allowing further progress. Duato was the first to propose a necessary and sufficient condition for deadlock-free routing in adaptive networks [3]. He noticed that alternative paths could be used to *escape* deadlock situations and that a cyclic dependency is not a sufficient condition to create a deadlock. He used Example 1, taken from [4], to demonstrate this.



(a) Interconnection network (b) Channel dependency graph

Example 1. Consider the interconnection network of Figure 1(a). Routing is defined as follows: when routing a packet from source n_i to destination n_j, the routing function always returns channel c_{Ai}. It returns channel c_{Hi} only if $j > i$. The c_{Hi} channels do not form a dependency cycle, implying that they will always eventually become available. The c_{Ai} channels do form a dependency cycle. However, even if all channels of this cycle are unavailable, messages in node n_0 can always *escape* the cycle by using channel c_{H0}. After this, the messages in the cycle can progress. For a more extensive explanation, we refer to Duato's book [4].

Based on his intuition, he defined and proved a condition capturing the fact that an adaptive network can still be deadlock-free even in the presence of cyclic dependencies between channels or buffers [3]. This was a breakthrough in the field as it enables a dramatic reduction in the number of resources to implement fully adaptive routing networks. It is also counterintuitive as it seems that deadlock may appear from a cyclic dependency. Duato's work was not easily accepted by his peers. On Duato's webpage one can read:

> Only a complex mathematical proof can show that deadlock freedom can be guaranteed if certain conditions are met. This research was so disruptive when it was developed that it was rejected by several peers and considered to be incorrect, even by the most prominent researchers at that time. However, it was finally accepted and several well-known researchers developed their own version of this theory.

The main contribution of this paper is the formal definition and proof of a necessary and sufficient condition for deadlock-free adaptive routing[1]. We express

[1] Sources available at
`http://www.cs.ru.nl/~julien/Julien_at_Nijmegen/ITP10-Pearl.html`

and prove this condition in the logic of the ACL2 theorem proving system [5]. This extends our formalization of Dally and Seitz' condition for deterministic routing [6]. In particular, the formalization of our condition makes use of second order functions and quantification. This illustrates the use of the *encapsulation* principle allowing the introduction of constrained functions and the use of the construct `defun-sk` allowing the introduction of universal and existential quantifiers. Our condition captures Duato's intuition of an *escape* more directly making the condition more intuitive. The proof of our condition is simpler than Duato's one. Nevertheless, our condition is logically equivalent as they are both necessary and sufficient conditions based on the same definition of a deadlock.

The paper is organized as follows. In the following Section, we present our condition. We provide our ACL2 formalization in Section 3 and give details on the proof in Section 4. Section 5 compares our condition and proof with those of Duato. We conclude in Section 6.

2 A New Necessary and Sufficient Condition

In this section we present a new necessary and sufficient condition for deadlock-free routing. Before defining our condition, we first provide a formal network model and a definition of a deadlock.

2.1 The Network Model

An interconnection network consists of processing nodes connected by channels. These nodes consist of *ports* and a central *switch* (see Figure 1). The switch contains the routing function and the switching method. There is a port for each in- and outgoing channel. Each node has a local in- and out-port, respectively for injecting and removing messages from the network. With each port a list - of size at least 1 - of *buffers* is associated. One buffer can store one packet. Bufferless switching can be represented by associating exactly one buffer per port. We assume that if a message is located in a buffer of its destination port, it is consumed immediately. Furthermore, we assume that all destination ports are *terminal*, i.e., they are not connected to other ports. A destination port is therefore never blocked.

With an interconnection network a *switching method* is associated. We consider store-and-forward packet switching (PS). Packets are the atomic objects

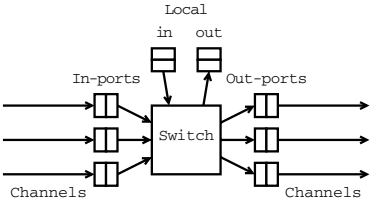


Fig. 1. Processing node, where each port has two buffers

transferred between ports. Each packet has a header storing its destination. Packets are stored in input buffers. Using the header information the routing function computes possible output buffers. A packet can make progress if one of these output buffers is free. Otherwise packets wait at their current position.

The *routing function* determines which paths a message can take from its current position to its destination. We define routing on the level of ports. Let P be the set of ports of the network. The routing function is defined as $\mathbf{R} : P \times P \mapsto \mathcal{P}(P)$, such that $\mathbf{R}(s, d)$ returns a set of ports which lead from port s to destination port d, i.e., the *next hops*.

The *port dependency graph* of routing function $\mathbf{R}$ – noted $G^R_{\mathtt{dep}}$[2] – is a graph with as vertices P and as edges the pairs of ports connected by function $\mathbf{R}$. Function $E_{\mathtt{dep}}$ returns the set of edges.

Given function $E_{\mathtt{dep}}$, we define the overloaded function $E_{\mathtt{dep}} : P \times P \mapsto P$ as follows:

$$E_{\mathtt{dep}}(p, d) \stackrel{\text{def}}{=} \{n \in E_{\mathtt{dep}}(p) \mid n \in \mathbf{R}(p, d)\}$$

When given extra parameter d, overloaded function $E_{\mathtt{dep}}$ returns a subset of the set of neighbors. $E_{\mathtt{dep}}(p, d)$ returns the set of dependency neighbors created by destination d, i.e., the set of next hops for a message located in p and destined for d.

2.2 Deadlock

A *configuration* σ is an assignment of packets to ports. A configuration is *legal* if and only if the buffer capacity of each port is not exceeded. A message is stuck if all its next hops are unavailable. A deadlock configuration is a configuration in which some messages are stuck forever. A canonical deadlock configuration is a configuration where *all* messages are stuck forever [4]. The canonical deadlock configuration can be obtained from a deadlock configuration by stopping the injection of new messages and waiting for arrival of all messages that are not blocked. We only need to consider canonical deadlock configurations [4]. From this point on, we define a *(canonical) deadlock configuration* to be a non-empty legal configuration where for all messages traversing the network there exists no available next hop. In such a configuration none of the messages can advance. A network is *deadlock-free* if and only if there does not exist a deadlock-configuration.

2.3 The Condition

Our condition uses the notion of *escape ports*. Given a set of ports P' a port $e \in P'$ is an escape for P' if and only if for all possible destinations there exists a dependency neighbor that is not contained in P':

$$\text{esc}(e, P') \stackrel{\text{def}}{=} \forall d \cdot E_{\mathtt{dep}}(e, d) \not\subseteq P'$$

[2] If the routing function is clear from the context we will omit it.

In other words, an escape for a subgraph is a port in which any message can be routed outside of the subgraph, regardless of the destination.

Fig. 2. Example dependency graphs. Ports A and B are the only destination ports. An edge labeled with d means that messages destined for d route to that neighbor. An unlabeled edge means that all messages route to that neighbor.

Example 2. Consider the set of black ports in Figure 2(a). This set has no escape: port C has no B-edge outside of the subgraph and likewise for port D and destination A. If all ports are full, all messages in C are destined for B, and all messages in D are destined for A, then the result is a deadlock-configuration. Figure 2(b) has an extra edge that makes port C an escape: for both destinations A and B there exists a neighbor outside the subgraph. There is no deadlock-configuration possible in this network.

If a port is in deadlock it must either be in some dependency cycle or there must exist a path that leads to some deadlocked cycle. Our condition is based on the idea that as long as dependency cycles have an escape, there is always a way to prevent the creation of deadlocks. Assume a cycle of unavailable ports, i.e., a circular wait. As long as such a cycle has an escape there is always at least one message with a next hop outside the cycle. If all sets of such cycles have an escape, it is always possible to prevent deadlocks to occur. Hence, the following condition:

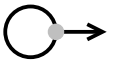
Theorem 1. *An interconnection network is deadlock-free if and only if all sets of cycles in the port dependency graph have an escape.*

Figures 3(a), 3(b), and 3(c) illustrate our condition. Assume a deadlock-free network such that its routing function has a cyclic dependency graph. Figure 3(c) shows the strength of the theorem. At some point, the escape of a cycle may lead to next hops which are included in cycles which have already been escaped. The theorem states that this set of cycles again has an escape, so that at least one message will eventually be able to escape this cycle of cycles.

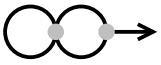
We rephrase Theorem 1 to a version which is logically equivalent.

Theorem 2. *An interconnection network is deadlock-free if and only if all subgraphs of the port dependency graph have an escape.*

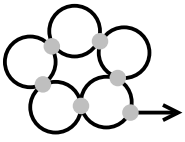
Although Theorem 1 is more intuitive, Theorem 2 is easier to formalize and to prove. In Section 3 we formalize Theorem 2 in the logic of the ACL2 theorem proving system.



(a) The escape leads messages out of the cycle.



(b) The escape leads to a cycle with a new escape.



(c) The escape leads to a cycle of cycles with a new escape.

3 Formalization

In this Section we show our formalization of the condition in the ACL2 theorem prover. First we explain how to formalize first- and second order theorems elegantly in ACL2. Then, the formalizations of routing functions, dependency graphs, deadlocks and escapes is provided. This section illustrates ACL2 features and therefore uses the LISP notion of ACL2.

3.1 ACL2

ACL2 stands for "A Computational Logic for Applicative Common Lisp". For an introduction to ACL2 we refer to [7]. The ACL2 logic is usually considered a quantifier-free first order logic with induction. The proofs in this paper require both quantifiers and second-order logic. ACL2 provides several mechanisms to elegantly introduce quantifiers and second order functions.

Quantifiers can be introduced using the `defun-sk` construct. This is a macro based on the construct `defchoose` for introducing quantified predicates. In this paper, we only consider the top-level construct `defun-sk`. We refer the reader to a nice introduction of `defchoose` by Ray [8].

The `defun-sk` construct introduces a skolemized function. To add a quantified formula to the ACL2 logic, one defines a skolemized function which returns `t` if and only if the formula holds. As an example, the formula $\exists x \cdot f(x, y) = 3$ can be added to the logic as follows:

```
(defun-sk formula (y)
  (exists (x) (equal (f x y) 3)))
```

A `defun-sk` construct can only introduce one quantifier. It adds to the ACL2 logic function `formula-witness`, which returns a witness for `x`. This witness function can then be used in further reasoning. Note that the actual witness is not explicitly computed and witness functions are not executable.

The `encapsulate` construct allows for second order universal quantification [9]. It introduces constrained functions. Those are functions with no actual definition but defined by a set of constraints. Theorems proven on these functions hold *for all* functions satisfying these constraints. Any function can be introduced, as long as a witness can be specified that satisfies the constraints.

To prove a certain property p for all commutative functions, one introduces function f and specifies a local witness as follows:

```
(encapsulate
  (((f * * => *))
   ;; Local witness:
   (local (defun f (a b) (equal a b)))
   ;; Constraint:
   (defthm f-commute (equal (f a b) (f b a)))))
```

No we can formulate the following second-order theorem:

```
(defthm ∀-commutative-f-holds-predicate-p
  (p (f a b)))
```

These techniques will be used extensively in the formalization and proof of our condition.

3.2 Routing and the Dependency Graph

Theorem 2 is a second order theorem ranging over all possible routing functions. Thus we define function **R** using the encapsulate construct. The only constraint on **R** is its typing: given a source and a destination port, **R** must return a non-empty set of valid routes. A route is valid if it is a path in the network starting with the source and ending with the destination. We define a straightforward recognizer function `Routesp` for valid routes and use it to enforce the typing of function **R**. For sake of readability we will omit the local witness needed for the `encapsulate` construct.

```
(encapsulate
 (((R * *) => *))
  (defthm Routesp-R
    (let ((routes (R p d)))
      (implies (not (equal p d))
               (and (consp routes)
                    (Routesp routes p d))))))
```

Regarding the dependency graph, we give function $E_{\mathtt{dep}}$ no explicit definition, but define it by two constraints. Constraint `routes-->edge` states that $E_{\mathtt{dep}}(p)$ must return at least all sets of next hops for all destinations reachable from p. Constraint `edge-->route` states that $E_{\mathtt{dep}}(p)$ must return at most those ports n for which there exists a destination d such that n is a next hop for a message destined for d.

```
(encapsulate
 (((E_dep *) => *))
  (defthm routes-->edge
    (implies (member-equal route (R p d))
             (member-equal (cadr route) (E_dep p))))
  (defthm edge-->route
    (implies (member-equal n (E_dep p))
             (find-dest-neighbor=next p n P))))
```

Function `find-dest-neighbor=next` searches through a set of possible destinations and returns a destination if and only if function $\mathbf{R}$ returns n as next hop for that destination.

```
(defun find-dest-neighbor=next (p n dests)
  (cond ((endp dests)
         nil)
        ((member-equal n (cadrs (R p (car dests))))
         (car dests))
        (t
         (find-dest-neighbor=next p n (cdr dests)))))
```

3.3 Deadlock

A message is stuck if all its next hops are unavailable. However, the exact definition of "being unavailable" depends on the switching technique implemented in the network. Furthermore, the exact and realistic definition of deadlock often depends on the underlying data-link protocol used to exchange messages between ports.

We want to abstract from these underlying mechanisms and we want to keep the proof as generic as possible. To these ends we define two application specific functions. These functions are not given a definition, but are genericly defined by constraints. First, function `unav` takes as parameters a port p and a configuration σ. It returns `t` if and only if p is unavailable for the respective switching method. Function $\forall$-`unav` gets as parameter a list of ports and returns `t` if and only if `unav` returns `t` for all these ports. Secondly, function `dl` returns `t` if and only if the given configuration is a legal deadlock for the respective switching method and data-link layer. The instantiation of function `dl` can be a complicated function, checking e.g. the states of buffers and signal spaces. We prove Theorem 2 for any switching method and data-link protocol such that function `dl` returns `t` if and only if there is no message with an available next hop.

To formalize this, we define function `find-free-msg` which searches through a list of messages and returns a message only if it has an available next hop. We define a constraint on function `dl` that it should return `t` if and only if function `find-free-msg` doen not return a message. The proof of the theorem can be applied for any combination of switching method and data-link layer for which this constraint can be discharged.

```
(defun find-free-msg (msgs σ)
  (cond ((endp msgs)
         nil)
        ((not (∀-unav (nexthops (car msgs))) σ)
         (car msgs))
        (t
         (find-free-msg (cdr msgs) σ))))
(encapsulate
  (((dl *) => *))
```

```
(defthm deadlock<->no-free-msg
  (iff (dl σ)
       (not (find-free-msg (msgs σ) σ)))))
```

Remark 1. This is not the only constraint on function `dl`. Otherwise, one could make a trivial instantiation by having functions `unav` and `dl` always return `nil`. Another constraint states that if `dl` returns `nil`, then the switching method must be able to advance at least one message. Advancement is expressed by a decreasing termination measure. For the formalization of this constraint, see [10].

3.4 The Condition

Theorem 2 ranges over both all possible configurations and all possible subgraphs of the port dependency graph. In ACL2, we elegantly define the theorem using the defun-sk construct.

Formalizing a function which returns `t` if and only if there is a possible deadlock configuration is straightforward:

```
(defun-sk ∃-deadlock ()
  (exists (σ)
      (dl σ)))
```

In order to formalize the right hand side of Theorem 2, we define function `escapep` which takes as parameters a port and a subgraph and returns `t` if and only if the given port is an escape for the subgraph. Function `∃-escape` returns `t` if and only if there is an escape for the given subgraph.

```
(defun-sk ∃-escape (subgraph)
  (exists (port)
          (escapep port subgraph)))
(defun-sk ∀-subgraphs-∃-escape ()
  (forall (subgraph)
          (implies (consp subgraph)
                   (∃-escape subgraph))))
```

Now the condition can easily be defined:

```
(defthm deadlock-free<->∀-subgraphs-∃-escape
  (iff (not (∃-deadlock))
       (∀-subgraphs-∃-escape)))
```

4 Proof

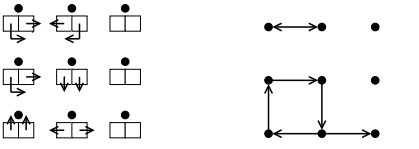
First we give a short informal proof sketch. Then we introduce definitions used in the proof and proceed with some details on the proof and its formalization.

We prove the contrapositive version of Theorem 2: there is a deadlock if and only if there is a subgraph without an escape. Such a subgraph is called a knot in a restricted dependency graph. We use two lemmas to build a knot from a

deadlock. The first lemma assumes a deadlock configuration σ. It states that if the waiting relation between the messages in σ is depicted in a graph, this graph contains a knot. This graph is called the *waiting graph*. The second lemma states that a knot in the waiting graph is also a knot in a *restricted dependency graph*. The third lemma creates a deadlock from such a knot. The three lemmas state the contrapositive of Theorem 2.

4.1 Definitions

The waiting graph is a graph which depicts the waiting relation between messages in a certain configuration. It is thus dynamically defined by a configuration σ. Informally, two ports p_0 and p_1 are connected in the waiting graph if there is message in a buffer of p_0 that is routed to p_1. Figure 3 gives an example.



(d) Configuration (e) Waiting graph

Fig. 3. An example configuration and its waiting graph. In the configuration each port has two buffers. Each arrow points to the next hop of the message in the buffer.

Definition 1. *Given a configuration σ, the* waiting graph *is defined by a set of vertices P and a set of edges $E^{\sigma}_{\texttt{wait}}$. There is an edge $(p_0, p_1) \in E^{\sigma}_{\texttt{wait}}$ if and only if there exists a message in a buffer of p_0 and p_1 is a next hop for that message.*

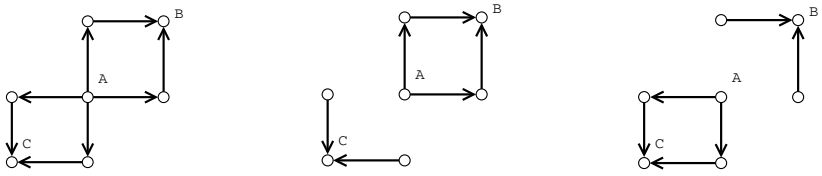
In the proof we consider subgraphs of the dependency graph defined by some restriction function $\delta : P \mapsto P$. This restriction function maps ports to destinations. At each port p, it restricts the dependency graph to edges leading to destination $\delta(p)$. Figure 4 gives an example.

Definition 2. *Given a restriction function $\delta : P \mapsto P$, the* δ-restricted dependency graph *is defined by a set of vertices P and a set of edges $E^{\delta}_{\texttt{dep}}$. There is an edge $(p_0, p_1) \in E^{\delta}_{\texttt{dep}}$ if and only if $p_1 \in E_{\texttt{dep}}(p_0, \delta(p_0))$.*

The proof uses the concept of *knots*. A knot in a graph is a subgraph where for each vertex all neighbors are included in the knot. Knots have proven to be a useful concept in deadlock detection [11].

Definition 3. *Given a graph G defined by functions V and E, a* knot *V' is a subgraph such that $\forall v \in V' \cdot E(v) \subseteq V'$.*

Given a graph G, function knot(P', G) returns `t` if and only if P' is a knot in graph G.



(a) Dependency graph (b) Restricted dependency graph (c) Restricted dependency graph

Fig. 4. Let B and C be the only destination ports. There are two δ-restricted dependency graphs: graph 4(b) is defined by $\delta(A) = B$ and $\delta(p) = E_{\texttt{dep}}(p)$ $(p \neq A)$; graph 4(c) is defined by $\delta(A) = C$ and $\delta(p) = E_{\texttt{dep}}(p)$ $(p \neq A)$.

4.2 The Proof

We prove that there is a deadlock if and only if there exists a δ-restricted dependency graph with a knot. Assume a deadlock configuration σ. We show that this deadlock-configuration implies a knot P' in the waiting graph of σ (Lemma 1). We then construct a restriction δ, such that P' is a knot in the δ-restricted dependency graph as well (Lemma 2). Assume a restriction δ and a knot P'. We can construct a deadlock configuration by filling all ports in P' with messages destined for the destinations provided by δ (Lemma 3). Hence, the contrapositive version of our theorem has been proven.

Lemma 1. *A deadlock configuration σ implies that there exists a knot P' in the waiting graph of σ.*

$$\exists \sigma \cdot \text{deadlock}(\sigma) \implies \exists P' \subseteq P \cdot \text{knot}(P', G^{\sigma}_{\texttt{wait}})$$

Proof. Take as P' the set of unavailable ports in σ. We prove that P' is a knot by contradiction. All the wait-neighbors are unavailable, since otherwise there would exist a port with a message with a next hop that is not full. This would imply that this message can move, which contradicts the assumption of deadlock. Since all wait-neighbors of P' are unavailable, and since P' is the set of all unavailable ports, P' is a knot. □

The deadlock does not necessarily consist of unavailable ports only: there can be a message in an available port, as long as each next hop for this message is unavailable. However, these available ports are not needed for the deadlock: the deadlock can be reduced to a set of unavailable ports only, while preserving the fact that each message is stuck.

Lemma 2. *A knot P' in the waiting graph of σ implies there exists a restriction δ such that P' is a knot in the δ-restricted dependency graph.*

$$\forall P' \subseteq P \cdot (\exists \sigma \cdot \text{knot}(P', G^{\sigma}_{\texttt{wait}}) \implies \exists \delta \cdot \text{knot}(P', G^{\delta}_{\texttt{dep}}))$$

Proof. Choose δ such that for all ports $p \in P'$, $\delta(p)$ returns the destination of one of the messages that is located in p in configuration σ. The set of neighbors created by this destination is a subset of the wait-neighbors of p – and thus also a subset of P' – since the wait-neighbors include all next hops created by the destinations of all messages in p. □

The reason why it is not possible to create a knot in the unrestricted dependency graph is that the messages in p do not necessarily lead to all possible reachable destinations. See for example Figure 5.

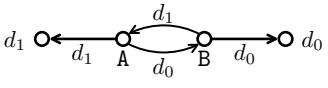


Fig. 5. Let A and B be filled with messages destined for respectively d_0 and d_1. Then $\{A, B\}$ is a knot in the waiting graph, but not in the unrestricted dependency graph.

Lemma 3. *A knot in the δ-restricted dependency graph implies a deadlock configuration.*

$$\exists P' \subseteq P \exists \delta \cdot \text{knot}(P, G^{\delta}_{\texttt{dep}}) \implies \exists \sigma \cdot \text{deadlock}(\sigma)$$

Proof. Construct a configuration σ by filling each port $p \in P'$ completely with messages destined for $\delta(p)$. The set of next hops of a message in p is the set of dependency neighbors created by destination $\delta(p)$. Since this is a subset of P' and since each port in P' is filled completely, all next hops of all messages are unavailable. Thus the configuration is in deadlock. □

Theorem 2 is the contrapositive of the existence of a knot in the δ-restricted dependency graph:

$$\exists P' \subseteq P \exists \delta \cdot \text{knot}(P, G^{\delta}_{\texttt{dep}}) \overset{\text{cp}}{=} \forall P' \subseteq P \exists e \in P' \cdot \text{esc}(e, P')$$

Thus a new necessary and sufficient condition for deadlock-free routing has been obtained: all subgraphs must contain an escape.

4.3 Details on the Formalization

We identify a peculiar aspect of the proof. The formalization of the right hand side of Lemma 2 has not been straightforward. The encapsulate construct enables us to define second order functions and prove universal theorems on them. However, Lemma 2 contains second order *existential* quantification. We formalized this by storing function δ in an association list. An association list is a common data type in LISP, which stores pairs of data. Each pair has a key and a value. An association list can therefore be used to store functions.

We define function E^{δ}_{dep}, the neighbor function of the restricted dependency graph as follows.

```
(defun E^δ_dep (port δ)
  (let ((dest (cdr (assoc port δ))))
    (filter-neighbors-for-dest port (E_dep port) dest)))
```

Given a port p, this function looks up the destination d associated with p using common LISP function `assoc`. It then filters all dependency neighbors to return only those that have been created by d. Function `restricted-dep-knotp` is a recognizer for a knot in the restricted dependency graph: it checks if for all ports the neighbors are contained in the given set of neighbors. If initially parameter `ports` is equal to parameter `neighbors`, then it returns `t` if and only if `neighbors` is a knot.

```
(defun restricted-dep-knotp (ports neighbors δ)
  (if (endp ports)
    t
    (and (subsetp (E^δ_dep (car ports) neighbors)
         (restricted-dep-knotp (cdr ports) neighbors)))
       ))
```

Finally, we introduce the existential quantifier of Lemma 2 using `defun-sk`.

```
(defun-sk E-restricted-dep-knot ()
  (exists (P' δ)
          (and (consp P')
               (restricted-dep-knotp P' P' δ))))
```

The proof of Lemma 2 requires a witness for function δ. This witness maps a port p to the destination of one of the messages in port p in σ. This witness can be built elegantly using the following function:

```
(defun build-δ-witness (ports σ)
  (if (endp ports)
    nil
    (acons (car ports)
      (dest-of (car (get-msgs-in-port (car ports) σ)))
      (build-δ-witness (cdr ports) σ))))
```

5 Comparison to Duato

Duato defined a necessary and sufficient condition for deadlock-free adaptive routing for PS [12]. We include this condition and a short clarification, but for an extensive explanation we refer to [4].

5.1 Duato's Condition

Theorem 3. *A connected and adaptive routing function R for an interconnection network I is deadlock-free if and only if there exists a routing subfunction R_1 that is connected and has no cycles in its extended resource dependency graph D_E.*

The intuition can be summarized as follows: assume a subgraph C_1 which contains all processing nodes, but contains only a subset of the channels of the network. Let C_1 satisfy two assumptions: (1) C_1 is acyclic and (2) the routing function is able to route any packet to any destination using channels in C_1 only. Then each message will always eventually reach its destination. Even if a message is stuck in a cycle, the channels of this cycle do not belong to C_1 by assumption (1). Assumption (2) states that for each node, any message can be routed to its destination through channels in C_1. Thus the message can always escape the cycles it is in by using channels in C_1.

Duato formalizes this notion using the concept of *routing subfunction*. Such a function only selects a subset of the possible next hops for each destination. This routing subfunction must be *connected*, i.e., able to route any packet to any destination. Furthermore, he extends the dependency graph with *direct cross dependencies*. If a packet stored in some channel could not have been routed to this channel by routing subfunction R_1, then dependencies involving this channel are direct cross dependencies. The *extended resource dependency graph* is the dependency graph with added direct cross dependencies.

5.2 Similarities and Differences

Our condition is logically equivalent to Duato's one. Both conditions are both necessary and sufficient for deadlock-free routing and the definitions of deadlock are equal. Furthermore, both conditions formalize the same intuition: there must always be an escape. Both conditions abstract from the data-link layer: they assume a message is stuck if and only if all its next hops are unavailable.

The differences lie in the formalization of the intuition. We straightforwardly formulate that there must always exist a message that is able to escape instead of stating that there must exist a routing subfunction capable to route messages through an acyclic subgraph. Moreover, the use of the regular dependency graph instead of the extended dependency graph reduces complexity.

The proofs are completely different. We prove the contrapositive form, namely that a deadlock is a subgraph without an escape. This enables a more constructive approach, since we merely had to construct a knot from a deadlock configuration. Duato constructs in his proof an acyclic connected routing subfunction from a network where no deadlock configuration is possible. This is the most difficult part of his proof.

6 Conclusion

We presented a necessary and sufficient condition for deadlock free adaptive networks. We formalized this condition and its proof using the ACL2 theorem

proving system. In particular, we illustrated existential quantification over constrained functions. This was elegantly performed using constructs of the ACL2 system specifically designed to extend its first order quantifier-free logic. The complete definition of our condition and its proof require about 100 "defuns" and 300 "defthms". Our proof is based on two different graphs. Each one of them needs its own definitions and lemmas. For instance, our formal proof states that the dependency graph really is the dependency graph of the given routing function. About 50 functions and 100 theorems are dedicated to define the dependency and waiting graphs. The main proof makes the connection between these graphs. To prove the existential quantifiers we need to define functions that build the deadlock configuration. These functions must be proven to produce a valid deadlock configuration. The proof follows the nice structure presented in Section 4 but many details are required to connect all the parts.

Our condition formalizes the seminal work of Duato. We provide a different condition and a different proof that both involve less concepts and are more intuitive. Nevertheless, both conditions are logically equivalent. The main advantage of our condition is that it has been fully verified using a mechanical theorem prover.

Acknowledgements. We would like to thank the anonymous reviewers for their apposite, constructive and detailed comments.

References

1. Stalling, W.: Operating Systems, Internals and Design Principles. Pearson Education International, London (2009)
2. Dally, W., Seitz, C.: Deadlock-free message routing in multiprocessor interconnection networks. IEEE Transactions on Computers (36), 547–553 (1987)
3. Duato, J.: A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. IEEE Transactions on Parallel and Distributed Systems 6(10), 1055–1067 (1995)
4. Duato, J., Yalamanchili, S., Ni, L.: Interconnection Networks, an engeneering approach. Morgan Kaufmann Publishers, San Francisco (2003)
5. Kaufmann, M., Manolios, P., Moore, J.S.: ACL2 Computer-Aided Reasoning: An Approach (2000)
6. Verbeek, F., Schmaltz, J.: Proof pearl: A formal proof of dally and seitz' necessary and sufficient condition for deadlock-free routing in interconnection networks. J. Autom. Reasoning (2009) (submitted to publication), http://www.cs.ru.nl/~julien/Julien_at_Nijmegen/JAR09.html
7. Kaufmann, M., Manolios, P., Moore, J.S.: ACL2 Computer Aided Reasoning: An Approach. Kluwer Academic Press, Dordrecht (2000)
8. Ray, S.: Quantification in Tail-recursive Function Definitions. In: Manolios, P., Wilding, M. (eds.) Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006), Seattle, WA, August 2006. ACM International Conference Series, vol. 205, pp. 95–98. ACM Press, New York (2006)
9. Kaufmann, M., Moore, J.S.: Structured Theory Development for a Mechanized Logic. Journal of Automated Reasoning 26(2), 161–203 (1997)

10. Verbeek, F., Schmaltz, J.: Formal specification of networks-on-chip: deadlock, livelock, and evacuation. In: Proceedings of Design, Automation & Test in Europe 2010 (DATE'10) (March 2010)
11. Misra, J., Chandy, K.: A distributed graph algorithm: knot detection. ACM Transactions on Programming Languages and Systems 4(4), 678–686 (1982)
12. Duato, J.: A necessary and sufficient condition for deadlock-free adaptive routing in cut-through and store-and-forward networks. IEEE Transactions on Parallel and Distributed Systems 7(8), 841–1067 (1996)

Extending Coq with Imperative Features and Its Application to SAT Verification*

Michaël Armand[1], Benjamin Grégoire[1], Arnaud Spiwack[2], and Laurent Théry[1]

[1] INRIA Sophia Antipolis - Méditerranée, France
{Michael.Armand,Benjamin.Gregoire,Laurent.Thery}@inria.fr
[2] LIX, École Polytechnique, France
Arnaud.Spiwack@lix.polytechnique.fr

Abstract. Coq has within its logic a programming language that can be used to replace many deduction steps into a single computation, this is the so-called reflection. In this paper, we present two extensions of the evaluation mechanism that preserve its correctness and make it possible to deal with cpu-intensive tasks such as proof checking of SAT traces.

1 Introduction

In the Coq proof assistant [3], functions are active objects. For example, let us consider the sum of two natural numbers. The sum function is defined recursively on its arguments using, say, Peano's definition. Then `1+2` is an expression that can be *computed* to its expected value. In particular to prove `1+2=3`, we simply need to know that equality is reflexive, and the system takes care of checking that `1+2` and `3` compute to the same value. Note that this computation (also called normalisation of λ-calculus) is not restricted to ground terms, like in our example: it can act as a symbolic evaluation on any term. Furthermore, Coq being based on the Curry-Howard isomorphism, writing a proof or a program is essentially the same. These remarks are the bases of proofs by reflection, which consist in replacing many deduction steps by a single computation. This technique has become popular in the Coq community since a few years. One of its most impressive application is the formal proof of the four-colour theorem [8].

Using reflection can greatly improve the checking time of proofs. However, as one pushes the limits of it, efficiency can become a concern. In that respect, a major improvement has already been achieved through the introduction of a dedicated virtual machine [9] allowing Coq programs to compare with (bytecode compiled) OCaml [12] ones. Still, there are strong restrictions left. First, there are no primitive data-structures. Every type is encoded using the constructs allowed by the system (primarily, inductive definitions). Also, there is no possibility to use destructive data-structures, which can be much more efficient than purely functional ones in some circumstances. To be able to go further on what can be efficiently programmed in Coq, we will add new data-structures such as

* This work was supported in part by the french ANR DECERT initiative.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 83–98, 2010.

native integers and destructive arrays. The challenge is to achieve this, changing as little as possible, in order to preserve trust in the correctness of evaluation in Coq, and nevertheless to get an effective speed-up thank to the new features.

The paper is organised in the following way. In Section 2, we describe how it is possible in Coq to benefit from the arithmetic capabilities of the microprocessor. Section 3 is dedicated to arrays. We then propose two examples that illustrate the benefit of our extension. In Section 4, we present the toy example of the Mini-Rubik for which we use computation to prove that it is always solvable in less than 11 moves. Section 5 is dedicated to a more challenging example. In order to prove the unsatisfiabily of large boolean propositions, we replay in a reflexive way proof traces generated by SAT solvers.

2 Extending Coq with Machine Integers

Arithmetic is currently defined in Coq as a standard inductive type. Thus, computations with numbers do not differ from other data-structures: it is a plain symbolic evaluation. What we aim at, here, is to rely on the arithmetic of the processor to speed-up computations within Coq. In order to add machine integers, a first possibility is to extend the theory underlying the Coq logic with:

- one primitive type int;
- the constructors $0, 1, 2, \ldots, 2^n - 1$ of type int;
- the basic primitive functions over the type int such as $+, *, \ldots$;
- the corresponding reduction rules for each primitive function.

It is also necessary to give it an equational theory, for instance, Peano theory together with a lemma stating that $(2^n - 1) + 1 = 0$. However, this approach has some drawbacks:

- It adds a large amount of new constructions to the theory. This goes against de Bruijn's principle which states that keeping the theory and its implementation as small as possible highly contributes to the trust one has in a system. Furthermore, on a more practical side, it will have a deep impact in the implementation, since the terms will have to be extended with new syntactic categories (primitive types and primitive functions).
- It adds a lot of new reductions, not only for ground terms but also for theorems. For example, if we consider the theorem n_plus_zero that states that $\forall n\colon \mathsf{int}.\, n + 0 = n$, it could be convenient to have (n_plus_zero 7) reduces to (refl int 7) where refl represents the reflexivity of equality. It is not clear that way that one captures all the necessary reductions.

For these reasons, we have taken an alternative approach. Efficient evaluation in Coq, as provided by the virtual machine, uses a compilation step. Before evaluating a term, it transforms it into another representation that is more suitable for performing reduction. The idea is to introduce the native machine integers not as part of the theory of Coq but only in this compilation phase. So, the type int is defined using the standard commands as a type with a single constructor that contains n digits:

Definition digit := bool.
Inductive int : Type := In(d_{n-1} ... d_1 d_0 : digit) : int.

Note that, in the current development, the type int is not parametric in n. We use a specific n to get a direct mapping to machine words and their operations. Still, the integration is done in a generic way so integers for a different n could be derived easily.

The primitive functions are not defined directly. We relate the machine numbers int with the relative numbers Z (the Coq representation of $\mathbb{Z}$) with the two functions $\overline{\bullet} : \mathsf{int} \to \mathsf{Z}$ and its inverse $[\bullet] : \mathsf{Z} \to \mathsf{int}$ and we prove that they satisfy the following two properties:

$$\begin{aligned} &\forall\, i : \mathsf{int}.\ [\overline{i}] = i \\ &\forall\, z : \mathsf{Z}.\ \overline{[z]} = z \mod 2^n \end{aligned}$$

Now, it is straightforward to define the primitive functions of int as the image of the corresponding function of Z. For example, addition for int is defined as follows:

Definition $i_1 +_{\mathsf{int}} i_2 := [\overline{i_1} +_{\mathsf{Z}} \overline{i_2}]$

It is also straightforward to derive the basic properties of these functions from the properties of the corresponding functions on Z. This set of definitions and properties will let the user manipulate the type int in Coq as any other type. So we preserve the property that everything is defined from base principle.

Now, the trick is to modify the compiler in such a way that it treats the type int as real machine integers. The main difficulty is that Coq requires strong reduction. This is not the case of traditional functional languages where only weak reduction is needed (no reduction under binders). Before explaining our modification to the compiler, we first give an overview of what symbolic weak and strong reductions are and then explain how the compiler works.

2.1 Strong Reduction by Symbolic Weak Reduction

In order to compute the strong normal form of a term t or to test the convertibility between two terms t_1 and t_2, the Coq system uses a compiled implementation of the symbolic calculus [9,2]. We briefly recall what symbolic computation is starting from the pure λ-calculus extended with inductive types.

Each inductive type is defined by a name I and a fixed number of constructors $|I|$. In this context the constructor In is represented by $C_{\mathsf{int},1}$. The syntax of the λ-calculus is extended with constructors and case analysis:

$$a ::= x \mid \lambda x.a \mid a_1\ a_2 \mid C_{I,i}(\boldsymbol{a}) \mid \mathsf{case}\ a\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow a_i)_{1\le i\le |I|}$$

the reduction rules are

$$\begin{array}{rcll} (\lambda x.a_1)a_2 & \Rightarrow & a_1\{x \leftarrow a_2\} & (\beta) \\ \mathsf{case}\ C_{I,j}(\boldsymbol{a})\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow a_i)_{1\le i\le |I|} & \Rightarrow & a_j\{\boldsymbol{x}_j \leftarrow \boldsymbol{a}\} & (\iota) \\ \Gamma(a) & \Rightarrow & \Gamma(a') \quad \text{if } a \Rightarrow a' & (\text{context}) \end{array}$$

where there is no restriction on the context Γ. Reduction can happen anywhere, in particular under binders or inside a branch of a case. We are interested in

computing the strong $\beta\iota$-normal form of the λ-term a. This is be done by iterating weak symbolic reduction and *readback*.

We first introduce the symbolic calculus:

$$
\begin{array}{lll}
\text{Extended terms} & b ::= x \mid \lambda x.b \mid b_1\ b_2 \mid C_{I,i}(\boldsymbol{b}) \mid \mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|} \mid [k] \\
\text{Accumulators} & k ::= h \mid k\ v \\
\text{Atoms} & h ::= \tilde{x} \mid \mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|} \\
\text{Values} & v ::= \lambda x.b \mid C_{I,i}(\boldsymbol{v}) \mid [k]
\end{array}
$$

The value $[h\ v_1 \dots v_n]$ is called an accumulator. It represents h applied to arguments $v_1 \dots v_n$. The atom $\tilde{x}$ is a symbolic variable. It represents the free variable x. Finally, $\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|}$ represents a suspended cases which cannot reduce since its argument does not reduce to a constructor.

The rules for weak reduction are defined as follows:

$$
\begin{array}{rcll}
(\lambda x.b)\ v & \to & b\{x \leftarrow v\} & (\beta_v) \\
[k]\ v & \to & [k\ v] & (\beta_s) \\
\mathsf{case}\ C_{I,j}(\boldsymbol{v})\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|} & \to & b_j\{\boldsymbol{x}_j \leftarrow \boldsymbol{v}\} & (\iota_v) \\
\mathsf{case}\ [k]\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|} & \to & [\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|}] & (\iota_s) \\
\Gamma_v(b) & \to & \Gamma_v(b') \quad \text{if } b \to b' & (\text{context}_v) \\
\multicolumn{4}{l}{\text{where } \Gamma_v(\bullet) ::= \bullet\ v \mid b\ \bullet \mid C_{I,i}(\boldsymbol{b} \bullet \boldsymbol{v}) \mid \mathsf{case}\ \bullet\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|}}
\end{array}
$$

The rules (β_v) and(ι_v) are the standard rules for call-by-value function application and case reduction. The rule (β_s) ("symbolic" β-reduction) handles the case where the function part of an application is not a function: a free variable $[\tilde{x}]$ or an application of a free variable $[\tilde{x}\ v_1 \dots v_n]$ or a suspended case. In that case, the accumulator simply absorbs its argument. The rule (ι_s) explains what to do when an accumulator is argument of a case: we simply *remember* that the case construct cannot reduce by producing a new accumulator. The rule (context_v) enforces weak reduction (no reduction under binder) and a right to left evaluation order (the argument being evaluated before the functional part)[1].

In order to compute the normal form of a λ-term a, we first inject a into the symbolic calculus. This is done by replacing each free variable x of a by its corresponding symbolic value $[\tilde{x}]$. We obtain a closed symbolic term: the variable $\tilde{x}$ is symbolic and not subject to substitution. In order to compute the normal form of a closed symbolic term b, we first compute its symbolic head normal form $\mathcal{V}(b)$ (also called value); then we *read back* the resulting value:

$$
\begin{array}{lll}
\mathcal{N}(b) & = \mathcal{R}(\mathcal{V}(b)) & (1) \\
\mathcal{R}(\lambda x.b) & = \lambda y.\mathcal{N}((\lambda x.b)\ [\tilde{y}]) \quad y \text{ fresh} & (2) \\
\mathcal{R}(C_{I,i}(\boldsymbol{v})) & = C_{I,i}(\mathcal{R}(\boldsymbol{v})) & (3) \\
\mathcal{R}([k]) & = \mathcal{R}'(k) & (4) \\
\mathcal{R}'(k\ v) & = \mathcal{R}'(k)\ \mathcal{R}(v) & (5) \\
\mathcal{R}'(\tilde{x}) & = x & (6) \\
\mathcal{R}'(\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|}) & = \mathsf{case}\ \mathcal{R}'(k)\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow \mathcal{N}(b\ C_i([\tilde{\boldsymbol{y}_i}])))_{1\le i\le |I|} & (7) \\
\multicolumn{3}{l}{\quad \text{where } b = \lambda x.\mathsf{case}\ x\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|}} \\
\multicolumn{3}{l}{\quad \text{and } \boldsymbol{y}_i \text{ are sequences of fresh variables with } |\boldsymbol{y}_i| = |\boldsymbol{x}_i|}
\end{array}
$$

[1] The evaluation order is important when using a virtual machine like the ZAM [11] with n-ary applications to execute the symbolic calculus.

The readback function $\mathcal{R}$ is defined recursively. It transforms a value v into a normalised source term. Reading back an atom $\tilde{x}$ (equation 6) simply consists in extracting the variable x. Reading back an accumulator $k\ v$ (equations 5) consists in applying the readback of the functional part to the readback of the argument. The interesting case is for function $\lambda x.b$ (equation 2). It consists in applying the functional value to a value $[\tilde{y}]$ representing a fresh variable. Here, "fresh" means that y is not a free variable of b. Then, we compute the value of the application, which reduces in one step to $b\{x \leftarrow [\tilde{y}]\}$, and reads it back as a normalised term a. The normal form of $\lambda x.b$ is $\lambda y.a$, which is correct up to α-conversion. The same idea is used to obtain the normal form of the branches of a case.

In [9], the authors prove the following theorem in the case of the λ-calculus:

Theorem 1. *If a is a closed, strongly normalizing λ-term, then $\mathcal{N}(a)$ is defined and is the normal form of a.*

The normal form of a term can be obtained by recursively computing its symbolic weak normal form and reading back the resulting value. The efficiency of the process clearly depends on the efficiency of the weak evaluation.

2.2 Compiling the Symbolic Calculus

Weak symbolic reduction can be implemented using a compiler and an abstract machine. The abstract machine we present here is a simplified version of the ZAM [11]. We write $\hat{v}$ the values manipulated by the abstract machine. They are pointers to heap allocated blocks $[T : \hat{v}_1, \ldots, \hat{v}_n]$, where T is a tag, and the $\hat{v}_i$ are values belonging to the block.

A machine state $(e,\ c,\ s)$ has three components: an environment e that contains a sequence of machine values $\hat{v}_1, \ldots, \hat{v}_n$ (it associates to the variable of de Bruijn index i the value $\hat{v}_i$); a code pointer c that represents the term being executed; a stack frame s that contains function arguments, intermediate results and return context $\langle c, e\rangle$. The semantics of the instruction set and the compilation rules are given Figure 1.

The compilation scheme $[\![b]\!]c$ takes a term b that has to be compiled and a code c that corresponds to the continuation of b. If b is normalising, the execution of $(e,\ [\![b]\!]c,\ s)$ leads to $(e,\ c,\ \hat{v} :: s)$ where $\hat{v}$ is the machine representation of the value v of b where the free variables have been substituted by their values in e.

Evaluating the code corresponding to a function $\lambda x.b$ builds a closure $[T_\lambda : c, e]$ where c is the code pointer corresponding to b and e the current environment. For application, a return context is pushed on the stack, then the argument and the function are evaluated, and finally, the APPLY instruction starts the evaluation of the closure. For constructors, the instruction MAKEBLOCK(n, T) builds a block[2] $[T : \hat{v}_1 \ldots \hat{v}_n]$ which is the machine representation of constructors. The compilation of a case starts by a PUSHRA, which saves the return context

[2] The compilation of a block erases the inductive name in the constructor. For the correctness, we refer to [9].

$$
\begin{aligned}
[\![x]\!]c &= \mathsf{ACCESS}(i);c \quad \text{where } i \text{ is the de Bruijn index of } x \\
[\![\lambda x.b]\!]c &= \mathsf{CLOSURE}(\mathsf{GRAB};[\![b]\!]\mathsf{RETURN});c \\
[\![b_1\ b_2]\!]c &= \mathsf{PUSHRA}(c);[\![b_2]\!][\![b_1]\!]\mathsf{APPLY} \\
[\![C_{I,i}(b_1,\ldots,b_n)]\!]c &= [\![b_n]\!]\ldots[\![b_1]\!]\mathsf{MAKEBLOCK}(n,i);c \\
[\![\mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\le i\le |I|}]\!]c &= \\
&\quad \mathsf{PUSHRA}(c);[\![b]\!]\mathsf{SWITCH}([\![b_1]\!]\mathsf{RETURN},\ldots,[\![b_{|I|}]\!]\mathsf{RETURN})
\end{aligned}
$$

$$
\begin{aligned}
(e,\ \mathsf{ACCESS}(i);c,\ s) &\rightsquigarrow (e,\ c,\ e[i]::s) \\
(e,\ \mathsf{CLOSURE}(c_f);c,\ s) &\rightsquigarrow (e,\ c,\ [T_\lambda : c_f, e]::s) \\
(e,\ \mathsf{GRAB};c,\ \hat{v}::s) &\rightsquigarrow (\hat{v}::e,\ c,\ s) \\
(e_f,\ \mathsf{RETURN},\ \hat{v}::\langle c,e\rangle::s) &\rightsquigarrow (e,\ c,\ \hat{v}::s) \\
(e,\ \mathsf{APPLY},[T_\lambda : c_f,e]::s) &\rightsquigarrow (e_f,\ c_f,\ s) \\
(e,\ \mathsf{PUSHRA}(c_1);c_2,\ s) &\rightsquigarrow (e,\ c_2,\ \langle c_1,e\rangle::s) \\
(e,\ \mathsf{MAKEBLOCK}(n,T);c,\hat{v}_1::\ldots::\hat{v}_n::s) &\rightsquigarrow (e,\ c,\ [T:\hat{v}_1::\ldots::\hat{v}_n]::s) \\
(e,\ \mathsf{SWITCH}(c_1,\ldots,c_m),\ [T:\hat{v}_1::\ldots::\hat{v}_n]::s) &\rightsquigarrow (\hat{v}_n::\ldots::\hat{v}_1::e,\ c_T,\ s) \\
(e,\ \mathsf{SWITCH}(c_1,\ldots,c_m),\ [0:\mathsf{ACCU},\hat{k}]::s &\rightsquigarrow (e,\mathsf{RETURN},\ \hat{v}::s) \\
\text{where } \hat{v} = [0:\mathsf{ACCU},[1:\hat{k},[T_\lambda:\mathsf{GRAB};\mathsf{SWITCH}(c_1,\ldots,c_m)]]] & \\
(e_f,\ \mathsf{ACCU},\hat{v}_a::\langle c,\ e\rangle::s) &\rightsquigarrow (e,\ c,\ [0:\mathsf{ACCU},e_f::\hat{v}_a]::s)
\end{aligned}
$$

Fig. 1. Compilation rules and semantics of the virtual machine

(used at the end of branches), then the argument is evaluated and the SWITCH instruction jumps to the corresponding branche.

What happens for the symbolic calculus? When an APPLY instruction is executed, the top stack value is not necessarily a closure, it can be the machine representation of an accumulator. An accumulator $[k]$ is represented like a closure: $[0:\mathsf{ACCU},\hat{k}]$. Furthermore, k being of the form $h\ v_1\ldots v_n$, $\hat{k}$ is represented as an environment: the sequence $\hat{h},\hat{v}_1,\ldots,\hat{v}_n$. The ACCU instruction takes the top value of the stack, pushes it at the end of the environment, rebuilds an accumulate block and returns. In that way, the APPLY instruction does not need to perform an extra test.

For the same reason, when a SWITCH instruction is executed, the top value is not necessarily a constructor, it can be an accumulator. If the tag is 0, the matched value is an accumulator, the SWITCH instruction builds an accumulate block representing the suspended case. In practice, 0 branches are automatically added to cases by the compiler, thus the SWITCH instruction of the ZAM can be used without extra test. Note that for atoms, $\hat{x}$ is represented by the block $[0:x]$ and $\mathsf{case}\ k\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\le i\le |I|}$ is represented by $[1:\hat{k},[T_\lambda:c,e]]$ where $\hat{k}$ is the machine representation of k, c and e are the code and the environment for the function $\lambda x.\mathsf{case}\ x\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\le i\le |I|}$.

In order to normalise a λ-term a using the virtual machine, we first compute $c = [\![a]\!]$ and start the evaluation with the abstract machine in the state $(e,\ c,\ \emptyset)$, where e is an environment associating to each free variable x of a its value $[\tilde{x}]$ encoded by the heap block $[0:\mathsf{ACCU},[0:\tilde{x}]]$. When the machine stops, we obtain a value v on the top of the stack. The readback function analyses which kind of value it is. It can either be a closure, a constructor, or an accumulator.

This can be done by a simple inspection of the tag. If the tag is T_λ, we have to normalise $v\ [\tilde{y}]$. This is done simply by restarting the machine in the state $(e, \mathsf{APPLY}, v :: [\tilde{y}] :: \emptyset)$. The same technique is used to normalise the branches of a suspended case.

2.3 Adding Machine Integers

We are now ready to explain how we can take advantage of the compilation mechanism to boost the evaluation of a λ-calculus extended with inductive types using the machine-integer operations. Of course, the gain will only be effective for programs using the previously defined inductive type int.

We extend the λ-calculus with a global environment Δ associating global variables g to their definition (λ-term):

$$a ::= x \mid \lambda x.a \mid a_1\ a_2 \mid C_{I,i}(\boldsymbol{a}) \mid \mathsf{case}\ a\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow a_i)_{1\le i\le |I|} \mid g$$
$$\Delta ::= \emptyset \mid \Delta :: (g, a).$$

The reduction rules now depend on the global environment Δ and are extended with one rule for the reduction of global definitions: $g \quad \rightarrow \quad \Delta(g)$.

We assume that n has been chosen in such a way that the term $\mathsf{In}(d_{n-1}, \ldots, d_0)$ is isomorphic to the machine word $d_{n-1} \ldots d_0$ if the d_j are all constructors (true stands for the machine digit 1 and false for 0). In the following, the term $\mathsf{In}(d_{n-1}, \ldots, d_0)$ is written p if all the d_j are constructors. We write $\dot{p}$ for the machine representation of p. If m is a machine integer, we write $|m|$ its representation as a term of type int; we have $p = |\dot{p}|$ and $m = |\dot{m}|$.

In the following, we assume that we have a global definition + performing the addition of two int. We denote by $+_a$ its associated definition and we write $+_M$ the processor addition. We assume that + does what it is supposed to do, i.e.:

$$p_1 + p_2 \quad \Rightarrow^* \quad |\dot{p}_1 +_M \dot{p}_1|$$

This gives us a first way to boost the reduction of + when the two arguments are of the form p_1 and p_2; instead of accessing to the global definition $+_a$ of + and then reducing the application $+_a\ p_1\ p_2$, one can directly compute $|\dot{p}_1 +_M \dot{p}_2|$. This solution does not work so well when additions are nested. For example, during the reduction of $(p_1 + p_2) + p_3$, the machine word $\dot{p}_1 +_M \dot{p}_2$ would be injected into its constructor representation at the end of the evaluation of $p_1 + p_2$ and then immediately re-injected into a machine word to perform the second addition. We have chosen a different solution that overcomes this problem.

We extend the symbolic calculus with machine integers and their primitive operations. The idea is to try to maintain as long as possible the terms of type int in their machine representation. The syntax of the new symbolic calculus is extended with machine integers:

$$\begin{aligned} b &::= x \mid \lambda x.b \mid b_1\ b_2 \mid C_{I,i}(\boldsymbol{b}) \mid \mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b_i)_{1\le i\le |I|} \mid [k] \mid m \\ v &::= \lambda x.b \mid [k] \mid C_{I,i}(v_1, \ldots, v_n) \mid m \\ \Delta_b &::= \emptyset \mid \Delta_b :: (g, v) \end{aligned}$$

In the definition of values, we exclude the case $\mathsf{In}(v_1, \ldots, v_n)$ where the v_i are all true or false. New reduction rules are added to the calculus. First, a special case is added for the constructor In:

$$p \quad \rightarrow \quad \dot{p}$$

In other words, when a constructor of type int can be represented by a machine word, its value is the machine representation. Second, for each global definition representing a primitive operation over int, some special rules are added. Let us consider addition, we add two rules:

$$\begin{array}{rcl} m_1 + m_2 & \rightarrow & m_1 +_M m_2 \\ v_1 + v_2 & \rightarrow & \Delta_b(+)\ v_1\ v_2 \end{array}$$

The first rule applies when the two arguments of $+$ are in machine representation. The result is given by the machine addition. The second one applies when one of the two arguments is not in machine representation. It can either be an accumulator or an In constructor with one of its arguments being an accumulator. In that case, the usual rule for global variable is used: the variable is replaced by is associating value in Δ_b.

Pattern matching on terms of type int has also to be taken care of. The matched value can be a machine word whereas a constructor value or an accumulator is expected. For this reason, we add the rule:

$$\mathsf{case}\ m\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1 \leq i \leq |I|} \quad \rightarrow \quad \mathsf{case}\ |m|\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1 \leq i \leq |I|}$$

Finally the readback function only needs to be extended so to get rid of machine integers: $\mathcal{R}(m) = |m|$.

Theorem 1 can be extended to this new symbolic calculus:

Theorem 2. *If for all global definitions g with a special shortcut we have $\Delta_b(g)\ \boldsymbol{m} \rightarrow^* g_M\ \boldsymbol{m}$. For all closed term a, well typed and strongly normalising, then $\mathcal{N}(a)$ is defined and is then normal form of a.*

What remains to be modified is the virtual machine and the compilation scheme. Previously, the values of the virtual machine were only pointers to heap-allocated blocks. The values are now extended with machine integers. Two instructions TOINT and OFINT are added to the virtual machine. Their semantics is given by:

$$\begin{array}{lrcll} (e, \mathsf{OFINT}; c, d_0 :: \ldots :: d_{n-1} :: s) & \rightsquigarrow & (e, c, m :: s)\ m = d_0 \ldots d_{n-1} & & (1) \\ (e, \mathsf{OFINT}; c, v_0 :: \ldots :: v_{n-1} :: s) & \rightsquigarrow & (e, c, v :: s) & \text{otherwise} & (2) \\ \text{where } v = [1 : v_0, \ldots, v_{n-1}] & & & & \\ (e, \mathsf{TOINT}; c, m :: s) & \rightsquigarrow & (e, c, v :: s) & & (3) \\ \text{where } v = [1 : d_0, \ldots, d_{n-1}] & & & & \\ (e, \mathsf{TOINT}; c, v :: s) & \rightsquigarrow & (e, c, v :: s) & \text{otherwise} & (4) \end{array}$$

We also add one instruction for each primitive operations. For example, the instruction ADD corresponds to the addition:

$$\begin{array}{lrcll} (e, \mathsf{ADD}; c, m_1 :: m_2 :: s) & \rightsquigarrow & (e, c, m_1 +_M m_2 :: s) & & (5) \\ (e, \mathsf{ADD}; c, v_1 :: v_2 :: s) & \rightsquigarrow & (e_+, c_+, v_1 :: v_2 :: \langle c, e \rangle :: s) & \text{otherwise} & (6) \\ \text{where } \Delta_b(+) = [T_\lambda : c_+, e_+] & & & & \end{array}$$

Finally, we modify the compiler with special cases for the compilation of the In constructor, for the primitive operations and for the pattern matching over elements of type int:

$$\begin{aligned}
[\![\mathsf{In}(b_0,\ldots,b_{n-1}]\!]c &= [\![b_{n-1}]\!]\ldots[\![b_0]\!]\mathsf{OFINT};c \\
[\![b_1+b_2]\!]c &= [\![b_2]\!][\![b_1]\!]\mathsf{ADD};c \\
[\![\mathsf{case}\ b\ \mathsf{of}\ (\boldsymbol{x}_i \Rightarrow b)_{1\le i\le |I|}]\!]c &= \\
\mathsf{PUSHRA}(c);&[\![b]\!]\mathsf{TOINT};\mathsf{SWITCH}([\![b_1]\!]\mathsf{RETURN},\ldots,[\![b_n]\!]\mathsf{RETURN}) \\
\text{if type of } b &= \mathsf{int}
\end{aligned}$$

The compilation of the In constructor generates an OFINT as last instruction and not a MAKEBLOCK as for the other constructors. The OFINT instruction checks if the first n arguments on the stack correspond to machine representation of digits (i.e. a block representing the constructors true or false). If all the arguments are constructors, the instruction builds the corresponding machine word (this corresponds to the reduction rule $p \to \dot{p}$ at the symbolic level). If one of the argument is not a constructor, the instruction is equivalent to $\mathsf{MAKEBLOCK}(n,1)$.

The compilation of a + first evaluates its arguments, then the ADD checks if they are machine words. If it is the case, the instruction simply performs the addition. If not, the instruction gets the value $[T_\lambda : c_+, e_+]$ of the +, inserts a return context $\langle c, e\rangle$ and performs an APPLY.

The compilation of a pattern matching on an object of type int is also modified. A TOINT instruction is inserted just before the SWITCH. If the top value of the stack is a machine word, the TOINT instruction replaces it by its corresponding block representation. If not, the TOINT instruction does nothing. The semantics of the SWITCH does not need to be modified.

The readback function should be able to analyse the value it gets. Before adding machine integer, this was done by matching the tag of the heap block. Remember that a heap block is a pointer, i.e. a machine integer. The problem is how to differentiate between pointers and integers. Note that a similar problem occurs for the implementation of the new instructions, which have to test if some values are blocks or integers. Fortunately, there is an easy solution. Since the implementation of the Coq virtual machine is based on the one of OCaml. The OCaml garbage collector makes the difference between a machine word representing a pointer and a machine word representing an integer using the following convention: a pointer is a machine word with least significant bit set to 0, an integer is a machine word with the bit set to 1. So, an integer p is encoded by the machine word $2p+1$. This is why OCaml, and now Coq, have integers of only 31 bits on a 32-bit architecture.

2.4 Primitive Functions

As adding a new primitive function requires some expertise, we have developed a reasonable library of primitive functions for the int type. It contains the usual functions (addition, multiplication, square root, comparison, logical functions, shifts) but also some iterators. Functions like

```
Definition foldi (A:Type) (F:A->A) (a:A) (n_s n_e:int) :=
  if n_s <= n_e then
     (fix aux (i:int) (ai:A) {
          if i = n_s then F i ai else aux (i-1) (F i ai)
      }) n_e a
  else a.
```

that computes $F\ n_s\ (F\ n_{s+1}\ (\ldots(F\ n_e\ a)\ldots))$ cannot be defined on top of our library. Because of the definition of int, this is not structurally recursive so Coq cannot establish that it always terminates. So, we add them as primitive functions.

3 Extending Coq with Persistent Arrays

Arrays are among the most important data-structures. Unfortunately, logics like the one of Coq are stateless. So, it is impossible to deal directly with destructive arrays as the ones we find in mainstream programming languages. The workaround is usually to use some flavour of purely functional arrays [13]. This works pretty well when arrays are rather small. For larger ones, not having an $O(1)$ access to elements of the array quickly becomes unmanageable.

In order to introduce destructive arrays, one way to go is to add states to the logic. Monads [18] are the standard way to do this. Unfortunately, monads are quite difficult to manage in a prover without developing some infrastructure (see [5,15] for example). An alternative approach is to develop some kind of program analysis that is capable of discovering (automatically or semi-automatically) that it can safely use destructive arrays instead of functional ones (see [14] for example). If one wants this technique to be applicable to a large set of programs, such analysis is usually rather complex. Here, we are going to follow a third approach and use destructive arrays but with a functional interface. These arrays are called persistent arrays [1]. For Coq, persistent arrays are implemented in a very naive way. An array is simply composed of the list of elements of the array and a default value.

Inductive array $(A : \mathsf{Type}) : \mathsf{Type}\ :=$ mkArray(elems : list A)(default : A).

As there is no exception in Coq, the default value is mainly used as a return value when accessing outside the range of the array. Instead of a default value, all functions manipulating arrays could have been parametrised by a proof that the access is valid. This last solution has two drawbacks. First, a proof has to be provided each time a function is used. Second in a call by value strategy, it adds extra costs since the proof has to be reduced before the actual function is evaluated. With a default value, the two basic operations on arrays are defined in a straightforward manner:

Definition get $(A : \mathsf{Type})(t : \mathsf{array}\ A)(n : \mathsf{int}) :=$ get_elem (default t) (elems t) n.
Definition set $(A : \mathsf{Type})(t : \mathsf{array}\ A)(n : \mathsf{int})(a : A) :=$
mkArray (upd_elem (elems t) $n\ a$) (default t).

where (get_elem $d\ l\ n$) returns the $n+1$-th element of the list l if n is less than the size of the list, d otherwise; and (upd_elem $l\ n\ a$) returns l where the

$n+1$-th element has been replaced by a if n is less than the size of the list, l otherwise. Both definitions are very inefficient. The access is linear in the number of elements and the update is also linear and furthermore reallocates a large part of the list.

Now, the virtual machine is going to conform to this functional behaviour but using destructive arrays. The idea is quite simple. Among all the versions of the array that may co-exist during execution, the last one (the newest one) is privileged and is represented by a destructive array. Look-ups and updates applied to this last version are then very efficient. Older versions of the array are not destructive arrays but point to the last version through a list of indirections. These indirections explain which modifications have to be applied in order to retrieve the values of the old array from the last version. Look-ups of old versions are possible (this is a requirement of the functional interface) but rather slow (linear to the number of updates). Updates just add a level on indirection. For the implementation, we have directly adapted the OCaml code proposed by J.C. Filliâtre in his paper [6]. Persistent arrays are defined as follows:

```
type 'a parray_kind = Array of 'a array | Updated of int * 'a * 'a parray
and 'a parray = ('a parray_kind) ref
```

A persistent array is a reference on a `parray_kind` which is either a destructive array (`Array`) or an indirection `Updated`(i,v,t) indicating that the persistent array is t except that at position i the value is v. The OCaml implementation does not contain explicitly the default value. It is stored in the last position of the array.

For the `get` function, we look directly in the array or follow the indirections:

```
let rec get p n =
  match !p with
  | Array t ->
     let l =  Array.length t in
     if 0 <= n && n < l then Array.get t n else Array.get t (l-1)
  | Updated (k,e,p) -> if n = k then e else get p n
```

Note that in a path to the destructive array, there could be several occurrences of `n` (this location could have been updated several times) but we stop at the first one. For the `set` function, an indirection is added at the right position:

```
let set p n e =
  let kind = !p in
  match kind with
  | Array t ->
     if 0 <= n && n < Array.length t - 1 then
        let res = ref kind in
        p := Updated (n, Array.get t n, res); Array.set t n e; res
     else p
  | Updated _ ->
     if 0<= n && n < length p then ref (Updated(n, e, p)) else p
```

Note that if we are updating outside the array, everything is left unchanged. Two more functions complement the library of arrays:

```
Definition copy (A : Type)(t : array A) := t.
Definition reroot (A : Type)(t : array A) := t
```

If the functional behaviour is the identity for both, they implement two distinct operations. With the first one, we get a physical copy of the array (a new independent destructive array is allocated). With the second one, we get an array with fast access (it is a destructive array). This is done without copying by recursively reverting all the indirections that lead to the destructive array in the array that is passed as argument. A very nice application of this reroot operation can be found in [6].

The extension of the virtual machine of Coq follows the same methodology than for machine integers. Two translation functions are used to transform a Coq representation of array into its virtual machine representation and conversely. The only detail we had to take care of is that OCaml arrays are limited in size. If the size of the Coq array is greater than the maximum OCaml size then the virtual machine switches to the inefficient Coq representation. For the compilation, array primitives like get are compiled in a slightly different way. This is due to the implicit polymorphism of the virtual machine implementation. For the virtual machine, the `get` operation expects only two arguments, whereas the Coq version expects three arguments (the type A of the elements). So, the compilation scheme first evaluates the last two arguments, then the `get` checks if they are in machine representation. If not, the argument A is evaluated and the three arguments are applied to the Coq implementation of get.

4 First Application: The Mini-Rubik

The Mini-Rubik is the pocket version of the famous Rubik cube. It is composed of 8 small cubes only and has 3,674,160 configurations. It is then quite easy to explore them completely with computers. Here, we explain how the property that the Mini-Rubik is always solvable in less than 11 moves has been proved formally.

First, we need to give a model. For this, we use indexation and associate to each configure of the Mini-Rubik a unique number from 1 to 3,674,160. Second, we have to construct a reachability graph – a Cayley graph, using the terminology of group theory. As we are capable of indexing configuration, this is easy. In order to represent a set of configurations, we use an array of 3,674,160 booleans. We prove that it is solvable in 11 moves using an iterative process and two sets of configurations S_A and S_N. The first set S_A contains the configurations that have been reached so far. The second set S_N contains the new configurations that have been reached by the previous iteration. Initially these two set only contain the initial configuration. At each iteration, S_A and S_N are updated with all the configurations that can be reached in one move by one configuration in S_N. After 12 iterations, S_N should be empty.

This application is perfect for testing our extension. First, as $3,674,160 < 2^{31} - 1$, a configuration can be represented by a single native integer. Second, if it is not possible to allocate in Coq an array of 3,674,160 booleans (booleans

are defined as an inductive type with two constructors, so one boolean takes one word in memory), thanks to binary encoding we use an array of machine integer of length $118522 = 3,674,160/31 + 1$. Furthermore, from a given configuration, there are 9 configurations reachable in one move. So this means that look-ups in the array S_A will be 9 times more frequent than updates. This is perfect since our look-ups cost much less than our updates. In [16], we have already presented a formal proof of the Mini-Rubik. In this version, machine integers were available but not efficient arrays. The arrays were implemented using an *ad hoc* functional data-structure. Checking the proof that the Mini-Rubik is solvable in 11 moves took 4 minutes. Modified with our new arrays, not only did it reduce to 10 seconds, but it also greatly simplified the implementation.

5 Second Application: Verifying SAT Traces

The most efficient SAT solvers that are used to prove the unsatisfiability of booleans formulas are all based on the DPLL algorithm with learning (see [10] for a complete introduction). An interesting feature of this algorithm is that very little overhead is needed in order to generate a trace that explains why the formula is unsatisfiable. Formats for traces may slightly vary from one solver to the other but they are all based on the simple resolution rule:

$$\frac{\neg x \vee C \qquad x \vee C'}{C \vee C'}$$

The variable x is called the resolution variable, C and C' are clauses, i.e. disjunctions of literals. The reflexive method to prove the unsatisfiability of boolean formulas in Coq works as follows:

- The initial problem is turned by some CNF transformation into a list of clauses. These clauses are called the roots
- The sat solver is called and returns the trace. This trace is composed of a list of resolution chains. Each chain corresponds to a clause that has been learned by the algorithm, so it is a logical consequence of the roots.
- A program written in Coq checks that the trace is correct: it builds the clauses that correspond to the resolution chains and finally checks that the last clause is the empty clause, i.e. $\bot$ is a consequence of the roots.

From the implementation point of view, roots are represented by a list of lists of natural numbers. Each boolean variable x has a unique number n. The literal x is represented as $2n$, $\neg x$ as $2n+1$. The trace is also represented as a list of lists of natural numbers. Each number represents an index of a clause. The indexes are computed with the following convention: roots come first then the clauses built by resolutions. A resolution chain is a list of natural numbers $\{n_1, n_2, \ldots, n_k\}$. In order to build the resulting clause, it is traversed from left to right: C_{n_1} is resolved with C_{n_2}, the result is then resolved with C_{n_3} and so on.

For the implementation of the checker, we have directly translated the C code of zVerify, the checker of zChaff [17], into our functional setting. The checker represents 363 lines of code and its correctness proof is 1621 line long. We use

Problem	Vars	Clauses	zChaff	Isabelle	Coq	Cert	Typing	Check	Array	Parray	zVerify
dubois50	150	400	0.00	0.04	0.04	0.00	0.02	0.02	0.00	0.00	0.01
barrel5	1407	5383	0.50	1.10	0.47	0.00	0.32	0.15	0.00	0.00	0.07
barrel6	2306	8931	1.74	10.38	1.15	0.08	0.62	0.45	0.06	0.14	0.14
barrel7	3523	13765	5.20	5.63	1.45	0.17	0.80	0.48	0.07	0.16	0.26
6pipe	15800	394739	42.21	–	24.73	0.98	13.92	9.83	2.05	4.74	2.86
longmult14	7176	22390	408.55	–	73.63	7.72	27.07	38.84	9.10	16.92	7.34
hole11	132	738	14.82	9.36	9.51	0.41	2.96	6.14	1.39	2.89	0.90
hole12	156	949	144.49	61.10	58.28	2.44	18.47	37.38	13.12	16.88	4.85
hole13	182	1197	5048.23	–	1068.30	88.15	387.44	592.72	183.47	275.14	–

Fig. 2. Benchmarking the checker

native integers to represent literals and indexes. Arrays are used for the set of clauses and for a temporary cache to compute the result of a resolution chain. In order to tackle large examples, we had to take a special care in memory usage. For this reason, traces are preprocessed for garbage collecting: we track when a clause is not used anymore, so its index can be reallocated and we share common prefixes in resolution chains. Traces processed by the checker are then list of tagged lists of natural numbers. The tag indicates if the new resulting clauses has to be appended or reallocated in that case it contains the index of the substituted clause.

In order to evaluate what we have done, Figure 2 presents some benchmarks. The machine used for these benchmarks is a Linux Intel Xeon 2.33GHz with 6144 KB of cache and 3 Gigabytes of memory. For each problem, we give:

- the number of variables, the number of clauses and the time for zChaff to generate the trace;
- the time of a very similar effort done in Isabelle/HOL by proof reconstruction [19];
- the time for the reflexive method in Coq (we first give the total time, and then split it in three: the time to parse and preprocess the trace, the time for Coq to typecheck the trace[3], and the actual time of the checker);
- the time of the extracted version of the checker running in OCaml with OCaml int and native compilation, first with destructive arrays (Array) then with persistent arrays (Parray);
- the time of zVerify.

Times are given in seconds. The symbol – indicates that the verification fails by out-of-memory. We can draw some observations. First, checking the trace is always faster than generating it. While we are slower than zVerify (roughly 10 times slower but with a better memory management most probably due to our preprocessing), we are competitive with the proof reconstruction of Isabelle/HOL. We have a better memory management. We are also faster for all benchmarks except the pigeon-hole problems, where proof reconstruction is as fast as our reflected approach. Second, a fair amount of time in Coq is spent for simply typechecking the generated trace. This clearly indicates that the typechecker of Coq has not yet been tuned to handle very large terms. Finally, the

[3] Coq has to typecheck the trace because it is an argument of the call to the checker, so it appears explicitly in the proof term.

checking part in Coq behaves quite well with respect to its extracted version with native code compilation. Remember that the evaluation in Coq is usually comparable to the bytecode compilation of OCaml and between native and bytecode compilations there is usually a factor of 5 to 10. One reason for this good behaviour is that our array primitives in Coq are actually running in their native version. The two versions of the extracted version indicate that, for this kind of application, the cost of using persistent arrays instead of destructive ones is about a factor of 2.

6 Conclusion

In this paper, we have presented how Coq can be extended with some imperative features. This extension increases the trusted computing base of the system but we believe that what we have proposed here is a very good compromise between the impact the extension has on the architecture of the prover and the benefit in term of speed-up in proof checking. Our changes are localised to the abstract machine and its compiler. We didn't have to change any other part of the prover. In particular, we didn't change the logic. We have also developed a systematic and simple methodology to add efficient data-structures with a functional interface to the abstract machine and its compiler. This contributes to the trust one can put in this extension. The methodology has been used to integrate machine integers and persistent arrays.

Some kind of destructive arrays are available in provers like ACL2 [4], Isabelle/HOL [5] or PVS [14], but some of these techniques are difficult to apply directly to a prover with a rich logic such as Coq and anyway all of them would require a major modification in the architecture of the prover. To our knowledge, the idea of using persistent arrays inside a prover is new. If it does not provide the full power of destructive arrays as in the other provers, for large applications, it gives a clear speed-up with respect to functional arrays. The loss in efficiency with respect to destructive arrays is largely compensated by the fact that we remain in the comfortable setting of functional behaviour.

Our overall goal is not to have an evaluation inside Coq that competes with mainstream programming languages. It is more to have a reasonable computing power within the prover. For example, being able to check the property of the Mini-Rubik in 4 minutes was sufficient enough. The SAT example is more interesting. We manage to get within Coq what was done by extraction in [7]. Without our extension, it would have been impossible to handle large examples. The fact that we could very quickly be competitive with what was achieved by finely-tuned proof reconstruction [19] in HOL and Isabelle/HOL is clearly good news. It opens new perspectives for the use of reflexive methods inside Coq. Finally, if our initial motivation was efficiency, memory usage has revealed to be sometimes an even more crucial limiting factor. Our machine integers and persistent arrays are much more compact than their corresponding functional representations – or their traditional encoding. Unfortunately, and this is maybe the only drawback of having this light integration, these objects only

exist within the abstract machine. In particular, they cannot be stored in proof objects, therefore have no impact on their size. For this reason, we had to develop an *ad hoc* inductive type in order to store efficiently the traces generated by the SAT solver.

References

1. Baker, H.G.: Shallow Binding Makes Functional Arrays Fast. ACM SIGPLAN notices 26, 145–147 (1991)
2. Barras, B., Grégoire, B.: On the Role of Type Decorations in the Calculus of Inductive Constructions. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 151–166. Springer, Heidelberg (2005)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
4. Boyer, R.S., Moore, J.S.: Single-Threaded Objects in ACL2. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 9–27. Springer, Heidelberg (2002)
5. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative Functional Programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)
6. Conchon, S., Filliâtre, J.-C.: A persistent union-find data structure. In: ACM Workshop on ML, pp. 37–46 (2007)
7. Darbari, A., Fischer, B., Marques-Silva, J.: Formalizing a SAT Proof Checker in Coq. First Coq Workshop, published as technical report tum-i0919 of the Technical University of Munich (2009)
8. Gonthier, G.: Formal Proof – The Four-Color Theorem. Notices of the AMS 55(11) (2008)
9. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP, pp. 235–246 (2002)
10. Kroening, D., Strichman, O.: Decision Procedures, An Algorithmic Point of View. Texts in Theoretical Computer Science. Springer, Heidelberg (2008)
11. Leroy, X.: The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA (1990)
12. Leroy, X.: Objective Caml (1997), `http://ocaml.inria.fr/`
13. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
14. Shankar, N.: Static Analysis for Safe Destructive Updates in a Functional Language. In: Pettorossi, A. (ed.) LOPSTR 2001. LNCS, vol. 2372, pp. 1–24. Springer, Heidelberg (2002)
15. Swierstra, W.: A Hoare Logic for the State Monad. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 440–451. Springer, Heidelberg (2009)
16. Théry, L.: Proof Pearl: Revisiting the Mini-Rubik in Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 310–319. Springer, Heidelberg (2008)
17. Princeton University. zChaff, `http://www.princeton.edu/~chaff/zchaff.html`
18. Wadler, P.: Monads for Functional Programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)
19. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. J. Applied Logic 7(1), 26–40 (2009)

A Tactic Language for Declarative Proofs*

Serge Autexier and Dominik Dietrich

German Research Center for Artificial Intelligence (DFKI), Bremen, Germany
{autexier,dietrich}@dfki.de

Abstract Influenced by the success of the MIZAR system many *declarative proof languages* have been developed in the theorem prover community, as declarative proofs are more readable, easier to modify and to maintain than their *procedural* counterparts. However, despite their advantages, many users still prefer the procedural style of proof, because procedural proofs are faster to write. In this paper we show how to define a *declarative tactic language* on top of a *declarative proof language*. The language comes along with a rich facility to declaratively specify conditions on proof states in the form of *sequent patterns*, as well as *ellipses* (dot notation) to provide a limited form of iteration. As declarative tactics are specified using the declarative proof language, they offer the same advantages as declarative proof languages. At the same time, they also produce declarative justifications in the form of a declarative proof script and can thus be seen as an attempt to reduce the gap between procedural and declarative proofs.

1 Introduction

The development of interactive tactic based theorem provers started with the LCF system [19], a system to support automated reasoning in Dana Scotts "Logic for Computable Functions". The main idea was to base the prover on a small trusted kernel, while also allowing for ordinary user extensions without compromising soundness. For that purpose Milner designed the functional programming language ML and embedded LCF into ML. ML allowed to represent subgoaling strategies by functions, called *tactics*, and to combine them by higher order functions, called *tacticals*. By declaring an abstract type *theorem* with only simple inference rules type checking guaranteed that tactics decompose to primitive inference rules.

While allowing for efficient execution of recorded proofs by representing them as a sequence of tactic applications, it has been recognized that these kind of proofs are difficult to understand for a human. This is because the intermediate states of a proof become only visible when considering the changes caused by the stepwise execution of the tactics. Tactic proofs can be extremely fragile, or reliant on a lot of hidden, assumed details, and are therefore difficult to maintain and modify (see for example [34] or [20] for a general discussion). As the only information during the processing of a proof is the current proof state and the next tactic to be executed, a procedural prover has to stop checking at the first error it encounters.

* This work was supported by German Ministry for Research and Education (BMBF) under grant 01 IW 07002 (project FormalSafe).

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 99–114, 2010.

This has led to *declarative proof languages*, inspired by MIZAR [29], where proof steps state *what* is proved at each step, as opposed to a list of interactions required to derive it. It has been argued that structured proofs in a declarative proof language are easier to read and to maintain. Moreover, as a declarative proof contains explicit statements for all reasoning steps it can recover from errors and continue checking proofs after the first error. It has been noted in [32] that a proof language can be implemented rather independently of the underlying logic and thus provides an additional abstraction layer. Due to its advantage many interactive theorem provers nowadays support declarative proofs (see for example [28,32,3,11]).

Another motivation for a declarative proof language comes from a research community dealing with the integration of proof assistants into development environments and supporting the so-called *document centric approach* [16,2]. The main idea is that the document, containing a formal theory and the formal proofs, is the central medium around which tools to assist the author are made available. As many tactics are developed while developing the formal theory, it is only consequent to integrate them into the document. Currently, this is not the case, as tactics are usually written in the underlying programming language of the prover.

Contributions. In this paper we present a *declarative tactic language* on top of a *declarative proof language* (which can be seen as an extension of [14]). To our best knowledge, such a language has not been presented before. Our language comes along with a rich facility to declaratively specify proof states (and conditions on them) in the form of *sequent patterns*, as well as *ellipses* (dot notation) to provide a limited form of iteration. The language is intended to provide a simple to use tactic language layer to bridge the gap between the predefined proof operators and the programming language of the proof assistant. This new layer of abstraction can be seen in analogy to what has been done by introducing declarative proof languages. We believe that declarative tactic languages offer similar advantages as declarative proof languages, namely robustness and readability, and that the trend towards declarative proof languages will carry on with declarative tactic languages. Interestingly, the trace of a declarative tactic is a declarative proof script and can thus be inserted into the document if desired. Moreover, because of its additional abstraction, it might provide possibilities to exchange reasoning procedures between different proof assistants in the long-term view.

The structure of the paper is as follows: Section 2 gives a more detailed background and motivates our language by means of a simple example. Section 3 presents the basic proof script language. Section 4 motivates and extends the language by an ellipsis construct. Finally, we conclude the paper in Section 5 with a discussion of related work.

2 Background and Introductory Example

At present, two main formalization styles are supported by interactive theorem provers, namely the *procedural style* and the *declarative style*.

In the procedural style, proofs consist of a sequence of tactic applications, as shown on the left of Figure 1. Intuitively, the proof script corresponds to the edges of a derivation tree being labeled with the tactic names. Even though there are different implementations of the notion of a tactic, each of these tactics can be understood as a program

```
theorem natcomp: "(a::nat) +
(b::nat) = (b::nat)+(a::nat)"
apply (induct a)
apply (subst add_0)
apply (subst add_0_right)
apply (rule refl)
apply (subst add_Suc_right)
apply (subst add_Suc)
apply (simp)
done
```

```
theorem natcomplus: "(a::nat) + b = b+a"
proof (induct a)
  show "0 + b = b + 0"
  proof (-)
    have "0+b=b" by (simp only: add_0)
    also have "...=b+0" by (simp only:
add_0_right)
    finally show ?thesis .
  qed
  next ...
```

Fig. 1. Procedural and declarative proof script in ISABELLE/ISAR

taking a list of goals together with a justification function as input and returning a new set of goals together with an updated justification function. The justification function is an internal function in the underlying programming language, such as ML, and cannot be presented to the user. By being a sequence of explicit program calls, a procedural proof contains explicit statements of *what to do*. In particular, the reader does not see the proof state unless he executes the tactic. Therefore, (procedural) proofs are considered not to be human readable and difficult to maintain. Procedural tactics are usually written in the underlying programming language of the assistance system, such as ML, conflicting with the document centric approach.

In the declarative style, proofs consist of structured blocks, where each block consist of a list of statements, connected by a fixed set of keywords. The statements specify what is proved at each step. Intuitively, a declarative proof script thus corresponds to the information contained in the nodes of a derivation tree. Most declarative languages require the user to give hints justifying the statement using previous statements. However, in principle a declarative proof can simply be a sequence of intermediate assertions, acting as islands or step stones between the assumptions and the conclusion (by omitting the constraints indicating how to find a justification of the proof step) leaving the task of closing the gaps to automation tools. Such islands are sometimes also called *proof plans* [13] or *proof sketches* [33]. Surprisingly, quite many systems – sometimes called *proof finders* or *proof planner* – have been developed trying to automatically close such gaps, such as MIZAR, NQTHM [8], the SPL system [34], the SAD system [30], the NAPROCHE [24] system, the SCUNAK system [9], TUTCH [1], or ΩMEGA [15].

The main advantage of a declarative proof script is that intermediate assertions are shown in the proof script. While this makes the proof easier to read, it makes it more difficult to write, as the proofs tend to be longer (see the example on the right of Figure 1). As a consequence, in practice users often prefer the procedural style of proof.

2.1 From Declarative Proof Scripts to Declarative Tactics

Having the correspondence between procedural and declarative proofs in mind and recalling that in the simplest case a tactic is a sequence of inference applications, representing a partial proof, it appears suggestive to think about declarative tactics as analog

```
theorem natcomplus: (a::nat) +
(b::nat) = b+a
proof
  subgoals by (induct a)
    subgoal 0 + b = b + 0
    subgoal Suc a + b = b + Suc a
      using IH:a+b=b+a
  end
qed
```

```
theorem natcomplus: (a::nat) +
(b::nat) = b+a
proof
  subgoals by (induct b)
    subgoal a + 0 = 0 + a
    subgoal a + Suc b = Suc b + a
      using IH: a+b=b+a
  end
qed
```

Fig. 2. Declarative proof with gap resulting by induction over a, respectively b

to procedural tactics. In contrast to procedural proofs the justification should be declarative, but we additionally require it to be specified using a declarative language, namely the proof language itself.

Consider for example the problem in Peano arithmetic of showing the commutativity of addition, that is, $a+b=b+a$. Of course, a proof can easily be generated in the procedural style. However, because of its advantages what we are really interested in is a declarative proof. Starting by induction over one variable, two possible proof attempts in ΩMEGA-proof language [15] are shown in Figure 2. Note that as the proof is still partial, it contains unjustified statements and can thus be seen as a proof sketch or a proof plan. This is similar to the "gap" command introduced in [16].

To automate the generation of one of these scripts, three steps are necessary. First, we need the *control information* over which variable the induction has to be performed. This is for example possible by analyzing the universally quantified variables and preferring those in recursion position. Second, we need a *schematic proof script* (a proof script with schematic variables), as well as a mechanism to instantiate schematic variables with actual terms, which is in our case the desired induction variable. Indeed, by comparing the scripts above, we observe that the two proof scripts can be made equal by replacing the induction variable by a schematic variable. The choice point over the induction variable can be expressed as membership in a (sorted) list of admissible induction variables, which can be easily computed in the underlying programming language of the prover. Finally, to be able to perform induction over the natural numbers on different problems, we replace the instance of the problem by a schematic variable, and use matching against the proof state to establish their relation.

```
strategy natinduct
 cases * ⊢ φ
    with x in (analyzeinductvars "φ")
      P=(abstract "φ" "x")  ->
proof
  subgoals by (induct x)
    subgoal P 0
    subgoal P (suc x) using IH: P x
  end
qed
```

Fig. 3. Declarative induction tactic

Our realization is shown in Figure 3, where we use quotes to refer to expressions in the tactic language within the underlying programming language. To illustrate the different levels of the tactic, we shade the background of expressions in the tactic language,

while expressions in the proof language are unshaded. Expressions in the underlying programming language are written in sans serif font.

Even though the result of executing a declarative proof script on a proof state is again a proof state – as in the case of procedural tactics –, it provides a simple mechanism to view the proofs at different levels of granularity, namely either as a single tactic invocation in the style of ... **by** `name`, but also by showing the declarative proof script obtained by replacing the schematic variables by the terms computed by the tactic. Note that one reason why some users favor the procedural style of proofs over the declarative style of proofs is that the procedural style is faster to type. This benefit remains when invoking declarative tactics. However, we additionally obtain a declarative proof script. In that sense, they can be seen as a means to close the gap between the procedural style and declarative style of proofs.

3 Development of the Language

In the simplest form, a declarative tactic is a (partial) proof in the proof language. For more complex situations, we have to answer the following questions: (i) When is the tactic applicable? (ii) How do the intermediate proof states (islands) look and how can they be generated? (iii) What is the justification for the statements?

To declaratively specify (i), we provide the **cases** construct and matching facilities to relate schematic variables with a given proof state and to restrict the applicability of the tactic (see Figure 4 for the grammar rules and Figure 3 for an example). The matcher specifies a matching condition on the proof state in the form of a sequent. * is used to indicate that the length of the antecedent of the sequent can be higher than the length of the antecedent of the matcher. **[***t***]** denotes the condition that *t* occurs as a subterm in a formula of the sequent. Subterm occurrences can further be restricted by the specification of polarities, where we use + to indicate a positive subformula and − to indicate a negative subformula. By using a schematic variable instead of +, −, the polarity is accessible within the tactic via that variable. Note that in general a matcher can match a given sequent in different ways and thus introduces nondeterminism.

defstrat	::= **strategy** *name stratexp*
stratexp	::= **cases** (*matcher stratexp*) $^+$
	| *proof* (**with** *assignments*)?
matcher	::= *matchhead whereexp*?
	(**with** *assignments*)
whereexp	::= **where** *prog*
matchhead	::= *sequent* | var
sequent	::= *termpatterns* (**,***)? ⊢ *termpattern*
termpatterns	::= *termpattern*
	| *termpatterns* **,** *termpattern*
termpattern	::= *form* | **[** *term* **]** (^ *termqualifier*)?
termqualifier	::= + | − | *var*
assignments	::= *lhs assignop prog*
assignop	::= **=** | **in**
lhs	::= *form* | **(** *form* (, *form*)$^+$ **)**

Fig. 4. Basic Tactic Language

proof	::= **proof** *steps* **qed**	*assume*	::= **assume** *steps from* **thus** *form*
steps	::= (*ostep*;*steps*)|*cstep*	*fact*	::= *sform* | *by from*
ostep	::= *set*|*assume*|*fact*|*goal*	*goals*	::= **subgoals** (*goal*)$^+$ *by*
cstep	::= *trivial* | *goals*|*cases*|ε	*cases*	::= **cases** (*form* { *proof* })$^+$ *by from*
by	::= **by** *name*? | *proof*	*goal*	::= **subgoal** *form* (**using** *form* (**and** *form*)$^+$)? *by*
from	::= **from** (*label* (, *label*)*)?	*set*	::= **set** *var*=*form* (**,** *var*=*form*)*
sform	::= *form* | **.** *binop form*	*trivial*	::= **trivial** *by from*

Fig. 5. ΩMEGA proof script language

(ii) is expressed within the proof language, while we allow the statements to contain schematic variables. Figure 5 shows the abstract syntax of our proof language, which is standard except that metavariables[1] are allowed in the statements. Metavariables can be instantiated using the **set** construct. The **subgoals** construct performs an explicit backward step. Each new subgoal is stated by a **subgoal**, followed by a proof of that subgoal. New assumptions for that subgoal are introduced within the **using** form. If only a single subgoal is introduced, the keyword **subgoals** can be omitted and the subsequent proofs refers to the justification of the reduction.

The value of schematic variables is computed during the expansion of tactic. The grammar rules for tactics are depicted in Figure 4. Here, *form* and *var* are from the underlying term language with possible labels on subterms, such as $(L1 : A) \wedge (L2 : B)$ and schematic variables. We use $\perp$ to indicate failure. To allow for a limited form of non-determinism we provide the assignment operator **in**, which chooses from a list of possibilities. As we cannot expect to provide a fixed language to express metalevel conditions and to perform metalevel analysis (in our case the extraction and sorting of the admissible induction variables), a reasonable strategy is to link-in the underlying programming language of the prover here, indicated in the grammar by *prog*.

For (iii), the justification is either underspecified (no **by** and no **from**), partially specified (**by** and/or **from**), or fully specified (subproof given).

These extensions already allow us the specification of the induction tactic in a declarative form (see Figure 3). Note that schematic variables can be used as placeholders for arbitrary terms, in particular terms generated by an oracle without justification. This provides a convenient means to integrate results from external systems, such as computer algebra systems (CAS) (see [10] for an overview for combining CAS systems and theorem provers). In such a case, we can either leave the justification of the oracle step underspecified (gap), or indicate a tactic to be used to justify it.

An example of such a tactic is given in Figure 6. The tactic is applicable if the goal has the form `abs(GOALLHS)<GOALRHS` and calls a CAS to factor `GOALLHS`. To that end, the schematic variable `Y` is bound to the result of the factorization provided by MAXIMA, where the translation of the term `GOALLHS` into the syntax of MAXIMA and the translation back is internalized in `maxima-factor`. If this succeeds, the script

[1] Metavariables are supported by the underlying ΩMEGA prover and are not to be confused with the schematic variables.

```
strategy maximafactorabs
cases
  * |- ((abs(GOALLHS)) < GOALRHS) ->
  proof
    subgoal abs(Y) < GOALRHS by
    proof
      L2:(Y = GOALLHS) by abeliandecide
      L3: abs(Y) = abs(GOALLHS) by (f=abs in arg_cong) from L2
      trivial from L3
    qed
  qed
  with Y = (maxima-factor "GOALLHS")
```

Fig. 6. Call of a CAS to factor a subterm of the goal formula

specified in the **proof** ... **qed** block is instantiated and inserted. Being executed, it reduces the goal `abs(GOALLHS)<GOALRHS` to the goal `abs(Y)<GOALRHS`. This reduction is justified by (1) showing the equality between `Y` (the factorization provided by the CAS) and `GOALLHS`, and then applying the fact that `abs` is a function. Note that the same tactic is also expressible in a forward style by relying on **assume** and that all labels in the proof script are generated at runtime and are renamed if already present in the context.

Semantics. Figure 7 shows the semantics of our language constructs by showing how to expand a declarative tactic to a declarative proof script. The expansion mechanism works on configurations $\langle PS; \Gamma; exp \rangle$, where PS denotes the current proof state, and Γ a context, which is initially empty and keeps track of bindings for schematic variables. exp denotes the expression to be expanded. Configurations evaluate either to a proof script, denoted by the relation $\hookrightarrow$, or to an environment, denoted by the relation $\rightarrow$. We use the notation $\Gamma \cup a = b$ to denote the update of Γ with the binding $a = b$, and the symbol $\perp$ to denote failure. To keep the rules simple, some rules are non-deterministic. In the actual implementation, of course, all results are lazily produced and stored for backtracking. $\mathsf{instance}(\Gamma, S)$ denotes the instantiation of the schematic proof S by replacing the schematic variables with their values in Γ. It is only applicable if all schematic variables are bound. We use eval to evaluate a LISP expression *prog*; the sequent matching is abstracted in the function match (which is also non-deterministic).

To enhance readability, we have grouped corresponding rules together. The first group describes the expansion of the **cases** construct, which returns the result of the first case that succeeds. An individual case is either a proof script (second group), or of the form *matchhead* (**where** *exp*)? (third group). The value of schematic variables is computed within the **with** construct (see the last group) which uses eval to evaluate expressions of the underlying programming language. Sequent matching works by first invoking the matcher on the current proof state and then evaluating additional condition.

$$\frac{\langle PS;\Gamma;c_1\rangle \hookrightarrow \bot \qquad \langle PS;\Gamma;\textbf{cases}\ c_2 \ldots c_n\rangle \hookrightarrow S}{\langle PS;\Gamma;\textbf{cases}\ c_1 \ldots c_n\rangle \hookrightarrow S} \qquad \frac{}{\langle PS;\Gamma;\textbf{cases}\ \varepsilon\rangle \hookrightarrow \bot}$$

$$\frac{\langle PS;\Gamma;c_1\rangle \hookrightarrow S}{\langle PS;\Gamma;\textbf{cases}\ c_1 \ldots c_n\rangle \hookrightarrow S}\ S \neq \bot$$

$$\frac{\langle PS;\Gamma;ass\rangle \to \bot}{\langle PS;\Gamma;proof\ \textbf{with}\ ass\rangle \hookrightarrow \bot} \qquad \frac{\langle PS;\Gamma;ass\rangle \to \Gamma' \qquad \langle PS;\Gamma';proof\rangle \hookrightarrow S}{\langle PS;\Gamma;proof\ \textbf{with}\ ass\rangle \hookrightarrow S}\ S \neq \bot$$

$$\frac{}{\langle PS;\Gamma;proof\rangle \hookrightarrow \mathsf{instance}(\Gamma, proof)}$$

$$\frac{\langle PS;\Gamma;matcher\rangle \to \bot}{\langle PS;\Gamma;matcher\ stratexp\rangle \to \bot} \qquad \frac{\langle PS;\Gamma;matcher\rangle \to \Gamma' \qquad \langle PS;\Gamma';stratexp\rangle \hookrightarrow S}{\langle PS;\Gamma;matcher\ stratexp\rangle \hookrightarrow S}$$

$$\frac{}{\langle PS;\Gamma;matchhead\rangle \to \mathsf{match}(matchhead, PS)} \qquad \frac{\langle PS;\Gamma;matchhead\rangle \to \bot}{\langle PS;\Gamma;matchhead\ \textbf{where}\ exp\rangle \to \bot}$$

$$\frac{\langle PS;\Gamma;matchhead\rangle \to \Gamma' \qquad \langle PS;\Gamma';\textbf{where}\ exp\rangle \to \bot}{\langle PS;\Gamma;matchhead\ \textbf{where}\ exp\rangle \to \bot}$$

$$\frac{\langle PS;\Gamma;matchhead\rangle \to \Gamma' \qquad \langle PS;\Gamma;\textbf{where}\ exp\rangle \to \Gamma''}{\langle PS;\Gamma;matchhead\ \textbf{where}\ exp\rangle \to \Gamma''}$$

$$\frac{\langle PS;\Gamma;ass\rangle \to \Gamma' \qquad \langle PS;\Gamma';\mathsf{eval}(c)\rangle \to \top}{\langle PS;\Gamma;\textbf{where}\ c\ \textbf{with}\ ass\rangle \to \Gamma'} \qquad \frac{\langle PS;\Gamma;ass\rangle \to \bot}{\langle PS;\Gamma;\textbf{where}\ c\ \textbf{with}\ ass\rangle \to \bot}$$

$$\frac{\langle PS;\Gamma;ass\rangle \to \Gamma' \qquad \langle PS;\Gamma';\mathsf{eval}(c)\rangle \to \bot}{\langle PS;\Gamma;\textbf{where}\ c\ \textbf{with}\ ass\rangle \to \bot} \qquad \frac{\langle PS;\Gamma;c\rangle \to \top}{\langle PS;\Gamma;\textbf{where}\ c\rangle \to \Gamma}$$

$$\frac{\langle PS;\Gamma;\mathsf{eval}(prog)\rangle \to c \qquad \langle PS;\Gamma \cup lhs = c;ass'\rangle \to \Gamma''}{\langle PS;\Gamma;lhs\ = prog\ ass'\rangle \to \Gamma''}\ c \neq \bot$$

$$\frac{\langle PS;\Gamma;\mathsf{eval}(prog)\rangle \to [c_1,\ldots,c_n] \qquad \langle PS;\Gamma \cup lhs = c_i;ass\rangle' \to \Gamma''}{\langle PS;\Gamma;lhs\ \textbf{in}\ prog\ ass'\rangle \to \Gamma''}\ n \geq 1 \wedge 1 \leq i \leq n$$

$$\frac{\langle PS;\Gamma;\mathsf{eval}(prog)\rangle \to \bot}{\langle PS;\Gamma;lhs\ assignop\ prog\ ass'\rangle \to \bot} \qquad \frac{\langle PS;\Gamma;eval(prog)\rangle \to c \quad \langle PS;\Gamma \cup lhs = c;ass'\rangle \to \bot}{\langle PS;\Gamma;lhs = prog\ ass'\rangle \to \bot}\ c \neq \bot$$

Fig. 7. Expansion Rules for a Declarative Tactic

4 Extension of the Basic Language

So far, our declarative tactic language is less expressive than its procedural counterpart. As a matter of fact, there exist powerful procedural tactics using for example the constructs of loops, such as simplification, which are per se difficult to express declaratively, as we cannot expect to determine their result unless we execute it. While this is unproblematic when using them to close gaps between intermediate statements, their treatment as black boxes makes it difficult to express how to process their results further in the form of a continuation, because the structure of the formula is lost. For example, all we know about the result term of factorization in Figure 6 is that it is of the form `Y`.

To illustrate a possible continuation of `maximafactorabs`, let us consider the so-called *limit domain* which contains statements about the limit and continuity of functions. It was proposed by Bledsoe [6] as challenge problems for automated theorem provers. The proofs typically involve ε-δ arguments and are interesting because both logic and computation have to be combined to find a solution to the problem given at hand, while still being simple enough to allow for an automatic solution based on heuristics. Several people from the AI community have tackled this domain (see for example [26, 4]) One heuristic of the limit domain to bound factors is to reduce the problem that the product $\beta\gamma$ is arbitrarily small to the problem that of showing that β is arbitrarily small and γ can be bounded. This heuristic is called *factor bounding* and described (in [4], p. 77f) as follows :

> "The following rule is stated for simplicity using only two factors, but the rule is implemented for a product of any number of factors.
>
> $$\frac{\Gamma, |\alpha| < \delta \qquad \Gamma, |\alpha| < \delta \vdash |\beta| < \varepsilon/(M+1)}{\Gamma, |\alpha| < \delta \vdash |\beta\gamma| < \varepsilon}$$
>
> When this rule is implemented, we take M to be a fresh metavariable, and forbid to M all the variables that are forbidden to δ. In the present implementation, the rule is used only when δ is a metavariable."

While our language can already deal with the factorization, we cannot yet declaratively express the factor bounding *for an arbitrary number of factors* within a single tactic and use it as continuation. Moreover, what we really want is to express the factor bounding as the continuation after successful factorization.

Reconsidering the problem we observe that the difficulty is due to the missing information we have about the factorization, namely the number of factors which is dynamic. Interestingly, even though the structure is dynamic, syntactic patterns are commonly used in mathematical practice to capture such a structure, by making use of ellipses (dot notation). In our example, a dynamic number of factors can be expressed by the pattern $X_1 * \ldots * X_n$. Internally, patterns are implemented by subsequently invoking the matcher for pattern X until it fails, taking the associativity of the binary operator into account, resulting in a list of matches which are stored in an internal variable X.

Coming back to our example, we notice that all factors but one factor shall be bounded. Therefore, we need also constructs to dynamically construct statements in the proof script language. To that end, we introduce a **foreach** construct. The grammar for the extended language constructs is shown in Figure 8. Binary patterns (*binop*) can be used in places where previously only *form* was allowed. Step is extended by *foreachstep* construct. Moreover, *assrhs* is extended by the foreach assignment (*foreachass*). These extensions will allow us to specify a variant of the factorbound method in a convenient way (see Figure 9 on page 109).

binoppat	::=	*pattern binop* **..** *binop pattern*
listaccess	::=	(*listterm* | *var*) _ (*var* | *number*)
listterm	::=	*listdel* **..** *listdel*
listdel	::=	*var* | *number* | *pattern*
pattern	::=	*binoppat* | *listaccess* | *from*
foreachexp	::=	**foreach** *var* **in** *listterm* (**where** cond)?
foreachstep	::=	*foreachexp* steps **end**
foreachass	::=	*foreachexp var* _ *var=prog*

Fig. 8. Dynamic matching constructs

Ellipses. So far our constructs for matching and constructing terms are static in the sense that their actual form was already determined at compile time. For example, a pattern of the form $lhs = rhs$ checks whether the input formula is an equality and binds its first argument to *lhs* and its right argument to *rhs*. *Dynamic Patterns* on the contrary are patterns that capture dynamic structures, such as all elements of a finite list. We support a simple dynamic pattern, an ellipsis for binary operators, written $A\ op \ldots op\ A'$, which acts like a Kleene star, as well as a list pattern which is similar except that *op* is omitted. Internally, such dynamic patterns are represented as lists, whose length is stored in an additional variable. To individually access the lists, we provide an accessor function _. That is, A_n denotes the n-th element in the list A. If n is a variable, then n is called *access variable*. In the current implementation, patterns are restricted to *simple patterns*, which are patterns that unify under a substitution σ whose domain consists only of access variables. Patterns can be used both in conditions, as left hand side of assignments, as well as in proof script terms. Some examples are shown in Figure 10.

The foreach construct provides a simple form of iteration over a list of values obtained from a dynamic pattern. It can be used to construct statements in the proof script language as well as to construct a list of schematic variables. Its expansion rules are shown in Figure 11, grouped into the expansion rules to expand **foreach** within a proof script, and the expansion rules to expand **foreach** in assignments. Note that in case of assignments a list containing all produced values is constructed, which has always the length of list over which it is iterated. In the case that the condition evaluates to $\bot$ a term *false* is inserted at the corresponding position.

Illustration of the Tactic. As an example, we consider the problem of proving $\lim_{x\to 3} \frac{x^2-5}{x-2} = 4$. After expanding the definition of lim, the proof state consists of the two goals $\varepsilon > 0, |x-3| < ?\delta \vdash |\frac{x^2-5}{x-2} - 4| < \varepsilon$ and $\varepsilon > 0 \vdash ?\delta > 0$. The declarative proof script is shown at the top of Figure 12, where the declarative tactic `factorbound` (see Figure 9) is not yet processed.

```
strategy factorbound
 cases
   abs(LHS)<RHS,* |- abs(GOALLHS) < GOALRHS
   where (and (variable-eigenvar.is "GOALRHS")
              (metavar-is "RHS")
              (some #'(lambda (x) (term= "LHS" "x")) "Y_1 .. Y_N"))
   with Y_1 * .. * Y_N = (maxima-factor "GOALLHS")
        j = (termposition "LHS" "Y_1 .. Y_N")
   ->
   proof
     L1: GOALLHS= Y_1 * .. * Y_N by abeliandecide
     foreach i in 1..N where (not (= "j" "i"))
       Y_j <= MV_j by linearbound
     end
     L2: abs(GOALLHS)=abs( Y_1 * .. * Y_N ) from L1
     .<= abs(Y_1) * .. * abs(Y_N)
     .< MV_1 * .. * MV_N
     .<= GOALRHS
   qed
   with foreach i in 1..N
        M_i = (if (= "i" "j") "RHS" (make-metavar (term-type "RHS")))
```

Fig. 9. Dynamic pattern matching and proof script generation

Expression	Meaning
`A_1 + .. + A_N`	finite sum with N summands
`abs(A_1) * .. * abs(A_N)`	product with N factors of the form `abs(_)`
`(X_1 + Y_1) * .. * (X_N + Y_N)`	product of terms of binary sums
`1 .. 5`	list [1,2,3,4,5]
`abs(A_1) .. abs(A_N)`	list with N terms of the form `abs(_)`

Fig. 10. Patterns using ellipses

Processing the `factorbound`-statement expands it and results in the following steps:

1. The pattern of the cases condition is matched, yielding the following binding: $\{\texttt{LHS} \mapsto x-3, \texttt{RHS} \mapsto ?\delta, \texttt{GOALLHS} \mapsto \frac{x^2-5}{x-2} - 4, \texttt{GOALRHS} \mapsto \varepsilon\}$
2. To be able to evaluate the **where** condition, the first **with** part is evaluated. This results in the following factorization: $Y_1 * \ldots * Y_n = (x-3) * (\frac{1}{x-2})(x-1)$. Internally, a list $Y = [(x-3), (\frac{1}{x-2}), (x-1)]$ is generated, n is bound to 3. In the next assignment, and j is bound to 1 by looking up $x-3$ in the list of factors.
3. The conditions of the where part evaluates to true
4. The **with** part of the **proof** is evaluated, generating a list $M = [?\delta, ?MV1, ?MV2]$ of length 3.
5. The **proof** part is expanded and inserted, resulting in the proof script shown at the bottom in Figure 12.

$$\frac{\langle PS;\Gamma;listterm\rangle \to [e_1,\ldots,e_n] \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n]\ (\textbf{where } c)^?\, exp_2\rangle \hookrightarrow S}{\langle PS;\Gamma;\textbf{foreach } var \textbf{ in } listterm\ (\textbf{where } c)^?\ exp_2 \textbf{ end}\rangle \hookrightarrow S}$$

$$\frac{}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } []\,(\textbf{where } c)^?\ exp_2 \textbf{ end}\rangle \hookrightarrow \varepsilon}$$

$$\frac{\langle PS;\Gamma\cup var = e_1;exp_2\rangle \hookrightarrow S1 \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n]\ exp_2\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n]\ exp_2\rangle \hookrightarrow S1\ S2}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma\cup var = e_1;exp_2\rangle \hookrightarrow S1\\ \langle PS;\Gamma\cup var = e_1;c\rangle \to \top\end{array} \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S1\ S2}$$

$$\frac{\langle PS;\Gamma\cup var = e_1;c\rangle \to \bot \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S2}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ exp_2\rangle \hookrightarrow S2}$$

$$\frac{\langle PS;\Gamma;listterm\rangle \to [e_1,\ldots,e_n] \qquad \langle PS;\Gamma;\textbf{iterate } ass \textbf{ in } [e_1,\ldots,e_n]\ (\textbf{where } c)^?\, prog\rangle \to \Gamma'}{\langle PS;\Gamma;\textbf{foreach } var \textbf{ in } listterm\ (\textbf{where } c)^?\ \underbrace{name\_var = prog}_{=:ass}\rangle \to \Gamma'}$$

$$\frac{\begin{array}{c}\langle PS;\Gamma\cup var = e_1;ass\rangle \to \Gamma'\\ \langle PS;\Gamma\cup var = e_1;c\rangle \to \top\end{array} \langle PS;\Gamma'\backslash(var = e_1);\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma''}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma''}$$

$$\frac{\langle PS;\Gamma\cup var = e_1;c\rangle \to \bot \qquad \langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_2,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma'}{\langle PS;\Gamma;\textbf{iterate } var \textbf{ in } [e_1,\ldots,e_n] \textbf{ where } c\ ass\rangle \to \Gamma'}$$

Fig. 11. Expansion of the **foreach** construct

Declarative Tactics and Parameters. For procedural tactics it is often convenient to pass control information in the form of arguments when calling the tactic. For example, in the introductory example we invoked the tactic `induct` with the argument "x" indicating the induction position. A similar mechanism is desirable in the case of declarative tactics. In our language, arguments are treated as schematic variables. If a schematic variable occurs in the proof script, but is neither used in the **cases** construct nor bound within the **with** environment, it corresponds to a required argument. Schematic variables that are computed within the tactic can be passed as optional arguments. In such a case, the passed argument overwrites the computed argument. We provide the common syntax *var*=*value* **in** *tactic*.

5 Conclusion and Related Work

In this paper we presented the construction a declarative tactic language on top of a declarative proof language. Our language comes along with a rich facility to declaratively specify proof states (and conditions on them) in the form of *sequent patterns*, as

```
theorem th1: lim_{x→3} (x^2−5)/(x−2) = 4
proof
  subgoals
    subgoal |(x^2−5)/(x−2) − 4| < ε using A1:ε > 0 and A2:|x−3| <?δ by factorbound
    subgoal ?δ > 0 using ε > 0
  end by limdefbw
qed
```

```
theorem th1: lim_{x→3} (x^2−5)/(x−2) = 4
proof
  subgoals
    subgoal |(x^2−5)/(x−2) − 4| < ε using A1:ε > 0 and A2:|x−3| <?δ
    proof
      L1:(x^2−5)/(x−2) − 4 = (x−3)∗((1/(x−2))∗(x−1)) by abeliandecide
      |x−1| ≤?MV1 by linearbound
      |1/(x−2)| ≤?MV2 by linearbound
      L2: |(x^2−5)/(x−2) − 4| ≤ |(x−3)∗((1/(x−2))∗(x−1))| from L1
      . ≤ |x−3|∗|1/(x−2)|∗|x−1|
      . <?δ∗?MV1∗?MV2
      . ≤ ε
    qed
    subgoal ?δ > 0 using ε > 0
  end by limdefbw
qed
```

Fig. 12. Declarative proof script of the example before and after processing the call of the declarative tactic `factorbound`

well as *ellipses* (dot notation) to provide a limited form of iteration. We believe that declarative tactic languages offer similar advantages than declarative proof languages, namely robustness, readability, and maintainability, because intermediate results of the tactic are visible due to the use of the declarative proof language for their specification. In addition to that, the main feature of declarative tactics is that they produce declarative proof scripts. They are thus a step to narrow the gap between the declarative and the procedural style, which is still frequently used in practice.

We have implemented 15 declarative tactics, all of which come in a variant that produces a forward style proof as well as in a variant that produces a backward style proof. So far the experiments confirm our impression that declarative tactics are well suited to automate (sub)proofs having a common structure, as is the case for induction proofs or the integration of external systems such as computer algebra systems. Moreover, operations that depend on the syntactic structure of the formula can easily be expressed, for example, to provide structure for common forms of forward reasoning. In these situations, the declarative tactics were easy to write. However, for situations in which the subsequent proof steps are not known in advance, such as simplification, declarative tactics are not adequate.

As already mentioned in the introduction, declarative proof languages and the verification of proof sketches has been studied by several people. There exists also several

approaches to present a machine-found proof in a user friendly way [21, 18]. In [27] a language is presented to automatically generate declarative proofs from proof terms. While this allows the presentation of proofs which have been found automatically, it does not deal with the specification of tactics in a declarative way.

Closely related to our work is ISAPLANNER [17]. ISAPLANNER generates proof plans and uses ISAR to represent them, that is, it also generates declarative proofs. It provides a "gap" command to represent open subgoals together with the annotation of a technique how to close such a gap. Compared to our approach, the main difference is that reasoning techniques are written as ML functions, whereas we use the underlying declarative proof language to specify the tactic. Moreover, our proof language differs from ISAR by allowing metavariables, which are not supported by ISAR, despite being supported by ISABELLE.

In the previous version of ΩMEGA, so-called *proof methods* were declaratively represented by *proof schemas*. Proof schemas were partial proofs in natural deduction (see [22]). In contrast to our approach, methods were implemented directly in the underlying programming language, no declarative proof language was used. Moreover, there was no possibility to pass control information in the form of a continuation.

Regarding intermediate tactic languages, our approach is similar to COQ's LTAC [12], which is an intermediate language intended to deal with small parts of proofs the user may like to automate locally. In contrast to our language, LTAC remains in the procedural style of the underlying tactic language instead of being declarative like our approach based on the declarative proof scripts. LTAC introduces conveniences of higher-level programming languages to the tactic script language which are independent form the underlying programming language and is similar in spirit to our aims. More specifically, LTAC provides pattern matching against the current goal, and our syntax for sequent patterns `|-` is inspired from it. LTAC also supports to match subterms and our syntax `[t]` is also the same here, except that we also allow to impose the polarity of the subformula supposed to match by `[t]+` or `[t]-`. A real extension of our language are the means to bind results of arbitrary computations to local script variables as well as the pattern syntax with ellipsis, which probably could be included in LTAC.

The matching part in case constructs of our tactic language is related to the extended meta-functions in ACL2 [23] which allow to access the current goal clause. The ACL2 meta-functions need to be proved correct in order to be usable by the ACL2 reasoner. From the LCF point of view, this is a way to include derived reasoning steps in the kernel proof rules, thus extending the kernel rules. In contrast to this our approach remains entirely in the LCF tradition since the declarative strategies generate proof scripts, which still need to be evaluated by the underlying (LCF-based) proof script interpreter. The possibility to perform arbitrary computations and bind the results to a term pattern, like the call to `maxima-factor` in the strategy *`factorbound`* is close to ACL2's `bind-free`, which takes an arbitrary binding list and adds it to the local context. This is also possible with our pattern approach by writing `X_1 .. X_N`, which has the advantage that the names of the local variables can be specified by the writer of the strategy. It would be possible to accommodate the `bind-free`-style in the pattern syntax, but so far we have not encountered situations where this was required. Moreover, the examples presented in [23] also bind only one variable.

In the context of rewriting several strategy languages exist. The general idea is to provide a language to specify a class of derivations the user is interested in by controlling the rule applications. Depending on the language, the language constructs are either defined by a combination of low-level primitives or build-in primitives. On a second layer, the languages provide constructs to express choice and sequencing, and recursion. Prominent examples are ELAN [7], MAUDE [25], and Stratego [31]. However, while being separate, these languages are not declarative in the sense that they are specified using a declarative language and produce declarative proofs.

References

1. Abel, A., Chang, B.-Y.E., Pfenning, F.: Human-readable machine-verifiable proofs for teaching constructive logic. In: Egly, U., Fiedler, A., Horacek, H., Schmitt, S. (eds.) Proceedings of the Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP '01), Universitá degli studi di Siena (June 2001)
2. Autexier, S., Benzmüller, C., Dietrich, D., Wagner, M.: Organisation, transformation, and propagation of mathematical knowledge in Omega. Journal Mathematics in Computer Science
3. Autexier, S., Fiedler, A.: Textbook proofs meet formal logic - the problem of underspecification and granularity. In: Kohlhase, M. (ed.) MKM 2005. LNCS (LNAI), vol. 3863, pp. 96–110. Springer, Heidelberg (2006)
4. Beeson, M.: Automatic generation of epsilon-delta proofs of continuity. In: Calmet, J., Plaza, J.A. (eds.) AISC 1998. LNCS (LNAI), vol. 1476, pp. 67–83. Springer, Heidelberg (1998)
5. Bertot, Y., Dowek, G., Hirschowitz, A., Paulin-Mohring, C., Théry, L. (eds.): TPHOLs 1999. LNCS, vol. 1690. Springer, Heidelberg (1999)
6. Bledsoe, W.W.: Challenge problems in elementary calculus. J. Autom. Reasoning 6(3), 341–359 (1990)
7. Borovansky, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with strategies in ELAN: A functional semantics. International Journal of Foundations of Computer Science 12(1), 69–95 (2001)
8. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Academic Press, London (1988)
9. Brown, C.E.: Verifying and invalidating textbook proofs using scunak. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 110–123. Springer, Heidelberg (2006)
10. Calmet, J., Homann, K.: Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In: Frontiers of Combining Systems (FroCos), pp. 221–234 (1996)
11. Corbineau, P.: A declarative language for the Coq proof assistant. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 69–84. Springer, Heidelberg (2007)
12. Delahaye, D.: A Proof Dedicated Meta-Language. In: Proceedings of Logical Frameworks and Meta-Languages (LFM), Copenhagen, Denmark, July 2002. ENTCS, vol. 70(2). Elsevier, Amsterdam (2002)
13. Dennis, L.A., Jamnik, M., Pollet, M.: On the comparison of proof planning systems: lambda-clam, Omega and IsaPlanner. Electr. Notes Theor. Comput. Sci. 151(1), 93–110 (2006)
14. Dietrich, D., Schulz, E.: Crystal: Integrating structured queries into a tactic language. J. Autom. Reasoning 44(1-2), 79–110 (2010)
15. Dietrich, D., Schulz, E., Wagner, M.: Authoring verified documents by interactive proof construction and verification in text-editors. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 398–414. Springer, Heidelberg (2008)

16. Dixon, L., Fleuriot, J.D.: A proof-centric approach to mathematical assistants. J. of Applied Logic: Towards Computer Aided Mathematics Systems 4(4), 505–532 (2005)
17. Dixon, L., Fleuriot, J.D.: Isaplanner: A prototype proof planner in Isabelle. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 279–283. Springer, Heidelberg (2003)
18. Fiedler, A.: *P.rex*: An interactive proof explainer. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 416–420. Springer, Heidelberg (2001)
19. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
20. Harrison, J.: Proof style. In: Giménez, E., Paulin-Mohring, C. (eds.) TYPES 1996. LNCS, vol. 1512, pp. 154–172. Springer, Heidelberg (1996)
21. Huang, X.: Human Oriented Proof Presentation: A Reconstructive Approach. DISKI, Infix, Sankt Augustin, Germany, vol. 112 (1996)
22. Huang, X., Kerber, M., Cheikhrouhou, L.: Adaptation of declaratively represented methods in proof planning. Annals of Mathematics and Artificial Intelligence 23(3-4), 299–320 (1998)
23. Hunt Jr., W.A., Kaufmann, M., Krug, R.B., Moore, J.S., Smith, E.W.: Meta reasoning in ACL2. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 163–178. Springer, Heidelberg (2005)
24. Kühlwein, D., Cramer, M., Koepke, P., Schröder, B.: The naproche system. In: Calculemus 2009, pp. 8–18 (July 2009)
25. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. In: Martí-Oliet, N. (ed.) Proceedings Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004), Barcelona, Spain. ENTCS, vol. 117, pp. 417–441. Elsevier, Amsterdam (2005)
26. Melis, E., Siekmann, J.H.: Knowledge-based proof planning. Journal Artificial Intelligence 115(1), 65–105 (1999)
27. Sacerdoti-Coen, C.: Declarative representation of proof terms. J. Autom. Reasoning 44(1-2), 25–52 (2010)
28. Syme, D.: Three tactic theorem proving. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 203–220. Springer, Heidelberg (1999)
29. Trybulec, A., Blair, H.: Computer assisted reasoning with MIZAR. In: Joshi, A. (ed.) Proceedings of the 9th Int. Joint Conference on Artifical Intelligence, M. Kaufmann, San Francisco (1985)
30. Verchinine, K., Lyaletski, A.V., Paskevich, A.: System for automated deduction (SAD): A tool for proof verification. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 398–403. Springer, Heidelberg (2007)
31. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001)
32. Wenzel, M.: Isar — a generic interpretative approach to readable formal proof documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–184. Springer, Heidelberg (1999)
33. Wiedijk, F.: Formal proof sketches. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 378–393. Springer, Heidelberg (2004)
34. Zammit, V.: On the implementation of an extensible declarative proof language. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 185–202. Springer, Heidelberg (1999)

Programming Language Techniques for Cryptographic Proofs*

Gilles Barthe[1], Benjamin Grégoire[2], and Santiago Zanella Béguelin[1]

[1] IMDEA Software, Madrid, Spain
{Gilles.Barthe,Santiago.Zanella}@imdea.org
[2] INRIA Sophia Antipolis - Méditerranée, France
Benjamin.Gregoire@inria.fr

Abstract. CertiCrypt is a general framework to certify the security of cryptographic primitives in the Coq proof assistant. CertiCrypt adopts the code-based paradigm, in which the statement of security, and the hypotheses under which it is proved, are expressed using probabilistic programs. It provides a set of programming language tools (observational equivalence, relational Hoare logic, semantics-preserving program transformations) to assist in constructing proofs. Earlier publications of CertiCrypt provide an overview of its architecture and main components, and describe its application to signature and encryption schemes. This paper describes programming language techniques that arise specifically in cryptographic proofs. The techniques have been developed to complete a formal proof of IND-CCA security of the OAEP padding scheme. In this paper, we illustrate their usefulness for showing the PRP/PRF Switching Lemma, a fundamental cryptographic result that bounds the probability of an adversary to distinguish a family of pseudorandom functions from a family of pseudorandom permutations.

1 Introduction

The goal of provable security [14] is to provide a rigorous analysis of cryptographic schemes in the form of mathematical proofs. Provable security holds the promise of delivering strong guarantees that cryptographic schemes meet their goals and is becoming unavoidable in the design and evaluation of new schemes. Yet provable security *per se* does not provide specific tools for managing the complexity of proofs and as a result several purported security arguments that followed the approach have been shown to be flawed. Consequently, the cryptographic community is increasingly aware of the necessity of developing methodologies that systematize the type of reasoning that pervade cryptographic proofs, and that guarantee that such reasoning is applied correctly. One prominent method for achieving a high degree of confidence in cryptographic proofs is to cast the security of a cryptographic scheme as a program verification problem: concretely, this is achieved by formulating security goals and

* This work was partially supported by French ANR SESUR-012, SCALP, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS, and Microsoft Research-INRIA Joint Centre.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 115–130, 2010.

hypotheses in terms of the semantics of probabilistic programs, and by defining the adversarial model in terms of a class of programs, e.g. probabilistic polynomial-time programs. The code-based approach leads to statements that are unambiguous and amenable to formalization. Yet, standard methods to verify programs (e.g. in terms of program logics) are ineffective to address the kind of verification goal that arises from cryptographic statements. The game-based approach [5,13] is an alternative to standard program verification methods that establishes the verification goal through successive transformations of the program and the goal. In a nutshell, the game-based approach proceeds by performing a sequence of transformations of the form $G, A \rightarrow^h G', A'$, where G and G' are probabilistic programs, A and A' are events, and h is a monotonic function such that $\Pr_G[A] \leq h(\Pr_{G'}[A'])$. When the security of a scheme is expressed as an inequality $\Pr_{G_0}[A_0] \leq p$, it can be proved by exhibiting a sequence of transformations $G_0, A_0 \rightarrow^{h_1} G_1, A_1 \rightarrow \cdots \rightarrow^{h_n} G_n, A_n$ and proving that $h_1 \circ \cdots \circ h_n(\Pr_{G_n}[A_n]) \leq p$.

CertiCrypt [3] is a fully machine-checked framework built on top of the Coq proof assistant [15] to help constructing and verifying game-based cryptographic proofs. An ancillary goal of CertiCrypt is to isolate and formalize precisely the reasoning principles that underlie game-based proofs and to automate their application. While many proof steps use standard reasoning principles based on observational equivalence and semantics-preserving program transformations, some essential techniques arise only in cryptographic proofs. The goal of this article is to present two such techniques and to illustrate their usefulness. The first technique is based on *failure events* and allows to formalize non-semantics-preserving transformations. It applies to transitions of the form $G, A \rightarrow^h G', A$, where G and G' behave identically unless a certain failure event bad occurs, and thus $h(p) = p + \Pr_G[\mathsf{bad}]$ (i.e. $\Pr_G[A] \leq \Pr_{G'}[A] + \Pr_G[\mathsf{bad}]$). The second technique uses interprocedural code motion to place upfront random choices made in games or, dually, to defer them until later in the game. These transformations, called *eager* and *lazy sampling* respectively, are widely used in proofs in the Random Oracle Model [4]. Both techniques are discussed in [3], but the present paper considerably extends their scope. Concretely, we complement the Fundamental Lemma of [3], that bounds the difference in the probability of events in two games that behave identically until failure, with a Failure Event Lemma that allows to bound the probability of failure. In order to establish the Failure Event Lemma, we introduce a notion of judgment that upper bounds the probability of an event after executing a program in terms of the probability of an event prior to execution. Moreover, we considerably clarify the eager/lazy sampling methodology using a logic for swapping program statements. The logic overcomes some limitations that hamper the application of the technique as it was described in [3].

Both extensions were required to prove the IND-CCA security of the OAEP padding scheme [8], for which the results of [3] did not suffice. As the complexity of this proof would prevent us from illustrating the techniques we used, we consider instead a simpler motivating example, namely the PRP/PRF Switching

Lemma, a fundamental cryptographic result that bounds the probability of an adversary to distinguish a family of pseudorandom functions from a family of pseudorandom permutations.

2 Motivating Example: The PRP/PRF Switching Lemma

Pseudorandom functions (PRF) and pseudorandom permutations (PRP) are two idealized primitives that play a central role in the design of symmetric-key systems. Although the most natural assumption to make about a blockcipher is that it behaves as a pseudorandom permutation, most commonly the security of a system based on a blockcipher is analyzed by replacing the blockcipher with a perfectly random function. The PRP/PRF Switching Lemma [10,5] fills the gap: given a bound for the security of a blockcipher as a pseudorandom permutation, it gives a bound for its security as a pseudorandom function.

Suppose you give an adversary black-box access to either a random function or a random permutation, and you ask her to tell you which is the case. For the sake of concreteness let us assume the domain of the permutation (and the domain and range of the function) is $\{0,1\}^\ell$. No matter what strategy the adversary follows, due to the birthday problem, after roughly $2^{\ell/2}$ queries to the oracle she will be able to tell in which scenario she is with a high probability. If the oracle is a random function, a collision is almost sure to occur, whereas it could not occur when the oracle is a random permutation. The birthday problem gives a lower bound for the advantage of the adversary. The PRP/PRF Switching Lemma gives an upper bound. In a code-based setting, its formulation is given in terms of two games G_{RP} and G_{RF}, that give the adversary access to an oracle that represents a random permutation and a random function, respectively:

Game G_{RP} :
$\boldsymbol{L} \leftarrow [\,];\ b \leftarrow \mathcal{A}()$
Oracle $\mathcal{O}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  $y \xleftarrow{\$} \{0,1\}^\ell \setminus \mathsf{ran}(\boldsymbol{L})$;
  $\boldsymbol{L} \leftarrow (x,y) :: \boldsymbol{L}$
return $\boldsymbol{L}(x)$

Game G_{RF} :
$\boldsymbol{L} \leftarrow [\,];\ b \leftarrow \mathcal{A}()$
Oracle $\mathcal{O}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  $y \xleftarrow{\$} \{0,1\}^\ell$;
  $\boldsymbol{L} \leftarrow (x,y) :: \boldsymbol{L}$
return $\boldsymbol{L}(x)$

where the instruction $y \xleftarrow{\$} \{0,1\}^\ell \setminus \mathsf{ran}(\boldsymbol{L})$ samples uniformly a bitstring of length ℓ that is not in the range of the association list $\boldsymbol{L}$, thus ensuring that oracle $\mathcal{O}$ implements an injective—and therefore bijective—function. Formally, the instruction $y \xleftarrow{\$} \{0,1\}^\ell \setminus \mathsf{ran}(\boldsymbol{L})$ may be implemented by repeatedly sampling a bitstring until the result does not belong to $\mathsf{ran}(\boldsymbol{L})$.

Lemma 1 (PRP/PRF switching lemma). *Let $\mathcal{A}$ be an adversary with black-box access to an oracle $\mathcal{O}$ implementing either a random permutation on $\{0,1\}^\ell$ as in game G_{RP} or a random function from $\{0,1\}^\ell$ to $\{0,1\}^\ell$ as in game G_{RF}. Suppose, in addition, that $\mathcal{A}$ makes at most $q > 0$ queries to its oracle. Then,*

$$|\Pr_{\mathsf{G}_{\mathsf{RP}}}[b=1] - \Pr_{\mathsf{G}_{\mathsf{RF}}}[b=1]| \leq \frac{q(q-1)}{2^{\ell+1}} \tag{1}$$

The standard *proof* of the PRP/PRF Switching Lemma is due to Impagliazzo and Rudich [10, Theorem 5.1]. Bellare and Rogaway [5] report a subtle error in the reasoning of [10] and provide a counterexample. They give a game-based proof of the PRP/PRF Switching Lemma under the additional assumption that the adversary never asks the same oracle query twice. Their proof uses the Fundamental Lemma (Sec. 4) to bound the advantage of the adversary by the probability of a failure event, but their justification of the bound on the probability of failure remains informal.

Shoup [13, Section 5.1] gives another game-based proof of the lemma under the assumption that the adversary makes exactly q distinct queries. In his proof, the challenger acts as an intermediary between the oracle and the adversary. Rather than the adversary calling the oracle at her discretion, it is the challenger who calls the adversary to get a query and who forwards it to the oracle. There is probably nothing wrong with this formulation, but we feel that it imposes unnecessary restrictions on the form of the adversary and hinders understanding.

The PRP/PRF Switching Lemma has been formalized previously. Affeldt, Tanaka and Marti [1] present a formalization of a game-based proof of the PRP/PRF Switching Lemma in Coq. What they prove in reality is a simplified variant that only holds for non-adaptive and deterministic adversaries. They formalize adversaries as purely deterministic mathematical functions that take a natural number and return an element in the domain of its oracle (a query). This implies that the queries the adversary makes do not depend on the responses to previous queries or on any random choices. The authors also report on a formalization in CertiCrypt [3]; Sec. 6 presents two significantly simplified proofs that use the programming language techniques developed in this paper.

3 A Language for Cryptographic Games

Games are formalized as programs in pWhile, a probabilistic imperative language with procedure calls. For the purpose of this exposition, we restrict random sampling to uniform distributions over a set $\mathcal{T}$ of base types. We let $\mathcal{V}$ be a set of variable identifiers[1] and $\mathcal{P}$ be a set of procedure identifiers. The set of commands is defined inductively by the clauses:

$$
\begin{array}{lll}
\mathcal{I} ::= & \mathcal{V} \leftarrow \mathcal{E} & \text{assignment} \\
\mid & \mathcal{V} \xleftarrow{\$} \mathcal{T} & \text{random sampling} \\
\mid & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
\mid & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{while loop} \\
\mid & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\
\mathcal{C} ::= & \mathsf{nil} & \text{nop} \\
\mid & \mathcal{I};\ \mathcal{C} & \text{sequence}
\end{array}
$$

[1] Variables are partitioned into local and global variables. We will sometimes ignore this distinction to avoid cluttering the presentation. We use a **bold face** to typeset global variables in games.

where we assume that the set $\mathcal{E}$ of expressions has been previously defined. Moreover, we assume given a function $\mathsf{fv} : \mathcal{E} \to 2^{\mathcal{V}}$ that computes the set of free variables of an expression.

A program (or game) consists of a command and an environment, which maps a procedure identifier to its declaration, consisting of its formal parameters, its body, and a return expression[2]

$$\mathsf{declaration} \stackrel{\text{def}}{=} \{\mathsf{params} : \mathcal{V}^*;\ \mathsf{body} : \mathcal{C};\ \mathsf{re} : \mathcal{E}\}$$

The language is deeply embedded in Coq, so one can define programs with holes by parametrizing them by program variables of type $\mathcal{C}$ (holes may appear in the main command or in the body of procedures in the environment). In particular, adversaries may be represented as procedures whose whole body is modeled as a variable of type $\mathcal{C}$.

In order to reason about games in the presence of unknown adversaries, we must specify an interface for adversaries, and construct proofs under the assumption that adversaries are well-formed against their specification. Assuming that adversaries respect their interface provides us with an induction principle to reason over all (well-formed) adversaries. We make an extensive use of the induction principle: each time a proof system is introduced, the principle allows us to establish proof rules for adversaries. Likewise, each time we implement a program transformation, the induction principle allows us to prove the correctness of the transformation for programs that contain procedure calls to adversaries.

Formally, the interface of an adversary consists of a triple $(\mathcal{F}, \mathcal{RW}, \mathcal{RO})$, where $\mathcal{F}$ is the set of procedures that the adversary may call, $\mathcal{RW}$ the set of variables that it may read and write, and $\mathcal{RO}$ the set of variables that it may only read. We say that an adversary $\mathcal{A}$ with interface $(\mathcal{F}, \mathcal{RW}, \mathcal{RO})$ is well-formed in an environment E if the judgment $\vdash_{\text{wf}} \mathcal{A}$ can be derived from the rules in Fig. 1. For convenience, we allow adversaries to call procedures outside $\mathcal{F}$, but these procedures must themselves respect the same interface. Note that the rules are generic, only making sure that the adversary makes a correct usage of variables and procedure calls. In particular, they do not aim to impose restrictions that are specific to a particular game, such as the number of calls that an adversary can make to an oracle, or the conditions under which an oracle can be called, or the computational complexity of the adversary. These additional assumptions on adversaries may be specified independently (e.g. by instrumenting games).

3.1 Semantics

Programs in pWhile are given a continuation-passing style semantics using the measure monad M, whose type constructor is defined as

$$M(X) \stackrel{\text{def}}{=} (X \to [0,1]) \to [0,1]$$

[2] The formalization uses single-exit procedures. For readability, all examples are presented in a more traditional style, and use an explicit return statement.

$$I \vdash_{\mathrm{wf}} \mathsf{nil} : I \qquad \frac{I \vdash_{\mathrm{wf}} i : I' \quad I' \vdash_{\mathrm{wf}} c : O}{I \vdash_{\mathrm{wf}} i; c : O} \qquad \frac{\mathsf{writable}(x) \quad \mathsf{fv}(e) \subseteq I}{I \vdash_{\mathrm{wf}} x \leftarrow e : I \cup \{x\}}$$

$$\frac{\mathsf{writable}(x)}{I \vdash_{\mathrm{wf}} x \xleftarrow{\$} T : I \cup \{x\}} \qquad \frac{\mathsf{fv}(e) \subseteq I \quad I \vdash_{\mathrm{wf}} c_i : O_i,\ i = 1, 2}{I \vdash_{\mathrm{wf}} \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : O_1 \cap O_2} \qquad \frac{\mathsf{fv}(e) \subseteq I \quad I \vdash_{\mathrm{wf}} c : I}{I \vdash_{\mathrm{wf}} \mathsf{while}\ e\ \mathsf{do}\ c : I}$$

$$\frac{\mathsf{fv}(\boldsymbol{e}) \subseteq I \quad \mathsf{writable}(x) \quad p \in \mathcal{F}}{I \vdash_{\mathrm{wf}} x \leftarrow p(\boldsymbol{e}) : I \cup \{x\}} \qquad \frac{\mathsf{fv}(\boldsymbol{e}) \subseteq I \quad \mathsf{writable}(x) \quad p \notin \mathcal{F} \quad \vdash_{\mathrm{wf}} p}{I \vdash_{\mathrm{wf}} x \leftarrow p(\boldsymbol{e}) : I \cup \{x\}}$$

$$\frac{\mathcal{RW} \cup \mathcal{RO} \cup \mathcal{A}.\mathsf{params} \vdash_{\mathrm{wf}} \mathcal{A}.\mathsf{body} : O \quad \mathsf{fv}(\mathcal{A}.\mathsf{re}) \subseteq O}{\vdash_{\mathrm{wf}} \mathcal{A}}$$

$$\mathsf{writable}(x) \stackrel{\text{def}}{=} \mathsf{local}(x) \vee x \in \mathcal{RW}$$

Fig. 1. Rules for well-formedness of an adversary against interface $(\mathcal{F}, \mathcal{RW}, \mathcal{RO})$. A judgment of the form $I \vdash_{\mathrm{wf}} c : O$ can be read as follows: assuming variables in I may be read, the adversarial code fragment c respects the interface, and after its execution variables in O may be read. Thus, if $I \vdash_{\mathrm{wf}} c : O$, then $I \subseteq O$.

The operators unit and bind of the monad are defined as follows:

$$\begin{array}{lll} \mathsf{unit} : X \to M(X) & \stackrel{\text{def}}{=} & \lambda x.\ \lambda f.\ f\ x \\ \mathsf{bind} : M(X) \to (X \to M(Y)) \to M(Y) & \stackrel{\text{def}}{=} & \lambda \mu.\ \lambda F.\ \lambda f.\ \mu(\lambda x.\ F\ x\ f) \end{array}$$

Expressions are deterministic; an expression e of type T is interpreted by a function $[\![e]\!] : \mathcal{M} \to [\![T]\!]$, where $[\![T]\!]$ is the interpretation of T. The denotation of a game G is given by the function $[\![G]\!] : \mathcal{M} \to M(\mathcal{M})$, that relates an initial memory $m \in \mathcal{M}$ to the expectation operator of the (sub) probability distribution of final memories resulting from its execution. This allows to define the probability of an event A in a game G and an initial memory m in terms of its characteristic function $\mathbb{1}_A$, as $\Pr_{G,m}[A] \stackrel{\text{def}}{=} [\![G]\!]\ m\ \mathbb{1}_A$. Thus, in this monadic semantics a probabilistic event is nothing but a continuation. We refer the interested reader to [3] for a more detailed account of the semantics.

3.2 Notations

Let X be a set of variables, $m_1, m_2 \in \mathcal{M}$ and $f_1, f_2 : \mathcal{M} \to [0, 1]$, we define

$$\begin{array}{rcl} m_1 =_X m_2 & \stackrel{\text{def}}{=} & \forall x \in X.\ m_1(x) = m_2(x) \\ f =_X g & \stackrel{\text{def}}{=} & \forall m_1\ m_2.\ m_1 =_X m_2 \implies f(m_1) = g(m_2) \end{array}$$

Let P be a predicate on X and let $\mu : M(X)$ be a distribution over X, then every value $x \in X$ with positive probability w.r.t μ satisfies P when

$$\mathsf{range}\ P\ \mu \stackrel{\text{def}}{=} \forall f.\ (\forall x.P\ x \implies f\ x = 0) \implies \mu\ f = 0$$

Our logics often use modify clauses; the statement $\mathsf{modify}(E, c, X)$ expresses that only variables in X are modified by the command c in environment E. Semantically,

$$\mathsf{modify}(E, c, X) \stackrel{\text{def}}{=} \forall m.\ \mathsf{range}\ (\lambda m'.\ m =_{\mathcal{V} \setminus X} m')\ ([\![E, c]\!]\ m)$$

Finally, for a Boolean-valued expression e, we let $\langle e \rangle_i$ denote the binary relation $\lambda m_1\ m_2.\ [\![e]\!]\ m_i = \mathsf{true}$.

3.3 Observational Equivalence and Relational Hoare Logic

CertiCrypt formalizes an equational theory of observational equivalence that allows to prove that program fragments are semantically equivalent. We say that two games G_1, G_2 are observationally equivalent w.r.t. an input set of variables I and an output set of variables O, when

$$\begin{array}{l}\vdash G_1 \simeq_O^I G_2 \stackrel{\text{def}}{=} \\ \quad \forall m_1\ m_2.\ m_1 =_I m_2 \implies \forall f_1\ f_2.\ f_1 =_O f_2 \implies [\![G_1]\!]\ m_1\ f_1 = [\![G_2]\!]\ m_2\ f_2\end{array}$$

Observational equivalence provides a useful tool to reason about probabilities. Assume that A is an event (i.e. a map from memories to Booleans) whose value only depends on a set of variables O, i.e. $\mathbb{1}_A =_O \mathbb{1}_A$. If $\vdash G_1 \simeq_O^I G_2$, then for every pair of memories m_1 and m_2 such that $m_1 =_I m_2$, we have

$$\Pr_{G_1,m_1}[A] = \Pr_{G_2,m_2}[A] \tag{2}$$

When $I = O = \mathcal{V}$, $\vdash G_1 \simeq_O^I G_2$ boils down to the semantic equivalence of both games, which we write as $G_1 \equiv G_2$.

Observational equivalence, however, is not enough to justify some context-dependent program transformations. In order to prove the correctness of such transformations, we need to generalize observational equivalence to a full-fledged Relational Hoare Logic that considers arbitrary binary relations on memories (and not just equality on a subset of variables). This logic deals with judgments of the form

$$\vdash G_1 \sim G_2 : \Psi \Rightarrow \Phi \stackrel{\text{def}}{=} \forall m_1\ m_2.\ m_1\ \Psi\ m_2 \implies [\![G_1]\!]\ m_1 \sim_\Phi [\![G_2]\!]\ m_2$$

where the relation $\sim_\Phi$ is a lifting of relation Φ to distributions, defined as:

$$\mu_1 \sim_\Phi \mu_2 \stackrel{\text{def}}{=} \exists \mu.\ \pi_1(\mu) = \mu_1 \wedge \pi_2(\mu) = \mu_2 \wedge \mathsf{range}\ \Phi\ \mu$$

where π_1 and π_2 are the projections that map a distribution over $A \times B$ to a distribution over A and B, respectively:

$$\pi_1(\mu) \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ (\lambda(x,y).\mathsf{unit}\ x) \qquad \pi_2(\mu) \stackrel{\text{def}}{=} \mathsf{bind}\ \mu\ (\lambda(x,y).\mathsf{unit}\ y)$$

For an overview of the rules of the relational logic we refer the reader to [3].

4 Failure Events

One common technique to justify a *lossy* transformation $G, A \to G', A$, where $\Pr_G[A] \neq \Pr_{G'}[A]$ is to annotate both games with a fresh Boolean flag bad that is set whenever the code of the games differ. Consider for example the following two program snippets and their annotated versions:

$$\begin{array}{lll} s \stackrel{\text{def}}{=} \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ c_2 & & s_{\mathsf{bad}} \stackrel{\text{def}}{=} \mathsf{if}\ e\ \mathsf{then}\ c_1; \mathsf{bad} \leftarrow \mathsf{true}; c\ \mathsf{else}\ c_2 \\ s' \stackrel{\text{def}}{=} \mathsf{if}\ e\ \mathsf{then}\ c_1; c'\ \mathsf{else}\ c_2 & & s'_{\mathsf{bad}} \stackrel{\text{def}}{=} \mathsf{if}\ e\ \mathsf{then}\ c_1; \mathsf{bad} \leftarrow \mathsf{true}; c'\ \mathsf{else}\ c_2 \end{array}$$

If we ignore the variable bad, s and s_{bad}, and s' and s'_{bad}, respectively, are observationally equivalent. Moreover, s_{bad} and s'_{bad} behave identically unless bad is set. Thus, the difference of the probability of an event A in a game G containing the program fragment s and a game G' containing s' instead can be bounded by the probability of bad being set in either s_{bad} or s'_{bad}, provided variable bad is initially set to false.

Lemma 2 (Fundamental Lemma). *For any pair of games G, G' and events A, A' and F:*

$$\Pr_G[A \wedge \neg F] = \Pr_{G'}[A' \wedge \neg F] \implies |\Pr_G[A] - \Pr_{G'}[A']| \leq \max(\Pr_G[F], \Pr_{G'}[F])$$

To apply the Fundamental Lemma, we developed a syntactic criterion to discharge its hypothesis for the particular case where $A = A'$ and $F = \mathsf{bad}$. The hypothesis can be automatically established by inspecting the code of both games: it holds if their code differs only after program points setting the flag bad to true, and bad is never reset to false afterwards. Note also that if both games terminate, then $\Pr_G[\mathsf{bad}] = \Pr_{G'}[\mathsf{bad}]$, and that if, for instance, game G' terminates with probability 1, it must be the case that $\Pr_G[\mathsf{bad}] \leq \Pr_{G'}[\mathsf{bad}]$.

4.1 A Logic for Bounding the Probability of Failure

Many steps in game-based proofs require to provide an upper bound for the expectation of some function g after the execution of a command c (throughout this section, we assume a fixed environment E that we omit from the presentation). This is typically the case when applying the Fundamental Lemma presented in the previous section: we need to bound the probability of the failure event bad (equivalently, the expected value of its characteristic function $\mathbb{1}_{\mathsf{bad}}$). An upper bound for a function $(\lambda m.[\![c]\!]\ m\ g)$ is a function f such that $\forall m.[\![c]\!]\ m\ g \leq f\ m$. We note this as a triple $[\![c]\!]g \preceq f$,

$$\vdash [\![c]\!]g \preceq f \stackrel{\text{def}}{=} \forall m.[\![c]\!]\ m\ g \leq f\ m$$

Figure 2 gathers some rules for proving the validity of such triples. The rule for adversary calls assumes that f depends only on variables that the adversary cannot modify directly (but that she may modify through oracle calls, of course). The correctness of this rule is proved using the induction principle for well-formed adversaries together with the rest of the rules of the logic.

The rules bear some similarity with the rules of Hoare Logic. However, there are some subtle differences. For example, the premises of the rules for branching statements do not consider guards. The rule

$$\frac{\vdash [\![c_1]\!]g \preceq f_{|e} \qquad [\![c_2]\!]g \preceq f_{|\neg e}}{[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]g \preceq f}$$

where $f_{|e}$ is defined as $(\lambda m.\ \text{if } [\![e]\!]m \text{ then } f(m) \text{ else } 0)$ can be derived from the rule for conditionals statements by two simple applications of the "rule of consequence". Moreover, the rule for conditional statements is incomplete: consider a

$$\vdash [\![\mathsf{nil}]\!] f \preceq f \qquad \frac{f = \lambda m.\ g(m\{x := [\![e]\!]\ m\})}{\vdash [\![x \leftarrow e]\!] g \preceq f} \qquad \frac{f = \lambda m. |[\![T]\!]|^{-1} \sum_{t \in [\![T]\!]}\ g(m\{x := t\})}{\vdash [\![x \xleftarrow{\$} T]\!] g \preceq f}$$

$$\frac{\vdash [\![c_1]\!] g \preceq f \quad [\![c_2]\!] h \preceq g}{\vdash [\![c_1; c_2]\!] h \preceq f} \qquad \frac{\vdash [\![c_1]\!] g \preceq f \quad [\![c_2]\!] g \preceq f}{\vdash [\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f} \qquad \frac{\vdash [\![c]\!] f \preceq f}{\vdash [\![\mathsf{while}\ e\ \mathsf{do}\ c]\!] f \preceq f}$$

$$\frac{\vdash g \leq g' \quad [\![c]\!] g' \preceq f' \quad f' \leq f}{\vdash [\![c]\!] g \preceq f} \qquad \frac{\vdash [\![p.\mathsf{body}]\!] g \preceq f \quad f =_X f \quad g =_Y g \quad x \notin (X \cup Y)}{\vdash [\![x \leftarrow p(e)]\!] g \preceq f}$$

$$\frac{\vdash_{\mathrm{wf}} \mathcal{A} \quad \forall p \in \mathcal{F}.\ \vdash [\![p.\mathsf{body}]\!] f \preceq f \quad f =_X f \quad X \cap (\{x\} \cup \mathcal{RW}) = \emptyset}{\vdash [\![x \leftarrow \mathcal{A}(e)]\!] f \preceq f}$$

$$\frac{f =_I f \quad \vdash c \simeq_O^I c' \quad g =_O g \quad \vdash [\![c']\!] g \preceq f}{\vdash [\![c]\!] g \preceq f}$$

Fig. 2. Selected rules of a logic for bounding events

statement of the form $[\![\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f$ such that $[\![c_1]\!] g \preceq f$ is valid, but not $[\![c_2]\!] g \preceq f$; the triple $[\![\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] g \preceq f$ is valid, but to derive it one needs to resort to observational equivalence. More general rules exist, but we have not formalized them since we did not need them in our proofs. More generally, it seems possible to make the logic complete, at the cost of considering more complex statements with preconditions on memories.

Discussion. The differences between the above triples and those of Hoare logic are inherent to their definition, which is tailored to provide an upper bound for the probability of an event after executing a command. Nevertheless, the validity of a Hoare triple $\{P\}c\{Q\}$ (in which pre and postconditions are Boolean-valued predicates) is equivalent to the validity of the triple $[\![c]\!] \mathbb{1}_{\neg Q} \preceq \mathbb{1}_{\neg P}$. There exists a dual notion of triple $[\![c]\!] g \succeq f$ whose validity is defined as:

$$\vdash [\![c]\!] g \succeq f \ \stackrel{\mathrm{def}}{=}\ \forall m. [\![c]\!]\ m\ g \geq f\ m$$

This dual notion allows to express termination of a program as $[\![c]\!] \mathbb{1}_{\mathsf{true}} \succeq \mathbb{1}_{\mathsf{true}}$. Moreover, there exists an embedding of Hoare triples, mapping $\{P\}c\{Q\}$ to $[\![c]\!] \mathbb{1}_Q \succeq \mathbb{1}_P$. The embedding does not preserve validity for non-terminating programs (under the partial correctness interpretation). Consider a program c that never terminates: we have $\{\mathsf{true}\}c\{\mathsf{false}\}$, but not $[\![c]\!] \mathbb{1}_{\mathsf{false}} \succeq \mathbb{1}_{\mathsf{true}}$ because for every $m \in \mathcal{M}$, we have $[\![c]\!]\ m\ \mathbb{1}_{\mathsf{false}} = 0$ and $\mathbb{1}_{\mathsf{true}}(m) = 1$.

4.2 Automation

In most applications of Lemma 2, the failure event can only be triggered by oracle calls. Typically, the flag bad that signals failure is set in the code of an oracle for which an upper bound for the number of queries made by the adversary is known. The following lemma provides a general method for bounding the probability of failure under such circumstances.

Lemma 3 (Failure Event Lemma). *Consider a game G that gives adversaries access to an oracle $\mathcal{O}$. Let P, F be predicates over memories, and let $h : \mathbb{N} \to [0,1]$ be such that F does not depend on variables that can be written outside $\mathcal{O}$, and for any memory m,*

$$\begin{aligned} P(m) &\implies \mathsf{range}\ (\llbracket \mathcal{O}.\mathit{body} \rrbracket\ m)\ (\lambda m'.\llbracket \mathsf{cntr} \rrbracket\ m < \llbracket \mathsf{cntr} \rrbracket\ m') \\ \neg P(m) &\implies \mathsf{range}\ (\llbracket \mathcal{O}.\mathit{body} \rrbracket\ m)\ (\lambda m'.F\ m' = F\ m \wedge \llbracket \mathsf{cntr} \rrbracket\ m = \llbracket \mathsf{cntr} \rrbracket\ m') \\ \neg F(m) &\implies \Pr_{\mathcal{O}.\mathit{body},m}[F] \leq h(\llbracket \mathsf{cntr} \rrbracket\ m) \end{aligned}$$

Intuitively, P indicates whether a call would increment the counter, failure F only occurs in calls incrementing the counter, and h bounds the probability of failure in a single call.

Then, for any initial memory m satisfying $\neg F(m)$ and $\llbracket \mathsf{cntr} \rrbracket\ m = 0$,

$$\Pr_{G,m}[F \wedge \mathsf{cntr} \leq q] \leq \sum_{i=0}^{q-1} h(i)$$

Proof. Define $f : \mathcal{M} \to [0,1]$ as follows

$$f(m) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \llbracket \mathsf{cntr} \rrbracket\ m > q \\ \mathbb{1}_F(m) + \mathbb{1}_{\neg F}(m) \sum_{i=\llbracket \mathsf{cntr} \rrbracket m}^{q-1} h(i) & \text{if } \llbracket \mathsf{cntr} \rrbracket\ m \leq q \end{cases}$$

We show $\llbracket G \rrbracket f \preceq f$ by structural induction on the code of G using the rules of the logic presented in the previous section. We first prove that $\mathcal{O}$ satisfies the triple $\llbracket \mathcal{O}.\mathsf{body} \rrbracket f \preceq f$; we must show that for every m, $\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ f \leq f(m)$. This is trivial when $\neg P(m)$, because we have

$$\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ f = f(m)\ (\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ \mathbb{1}_{\mathsf{true}}) \leq f(m)$$

When $P(m)$ and $\llbracket \mathsf{cntr} \rrbracket\ m \geq q$, this is trivial too, because $\mathcal{O}.\mathsf{body}$ increments cntr and the left hand side becomes 0. We are left with the case where $P(m)$ and $\llbracket \mathsf{cntr} \rrbracket\ m < q$. If $F(m)$, the right hand side is equal to 1 and the inequality holds. Otherwise, we have from the hypotheses that

$$\begin{aligned} \llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ f &\leq \llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ \left(\lambda m'.\mathbb{1}_F(m') + \mathbb{1}_{\neg F}(m') \sum_{i=\llbracket \mathsf{cntr} \rrbracket m'}^{q-1} h(i) \right) \\ &\leq \Pr_{\mathcal{O}.\mathsf{body},m}[F] + \llbracket \mathcal{O}.\mathsf{body} \rrbracket\ \left(\lambda m'.\mathbb{1}_{\neg F}(m') \sum_{i=\llbracket \mathsf{cntr} \rrbracket m+1}^{q-1} h(i) \right) \\ &\leq h(\llbracket \mathsf{cntr} \rrbracket m) + (\llbracket \mathcal{O}.\mathsf{body} \rrbracket\ m\ \mathbb{1}_{\neg F}) \sum_{i=\llbracket \mathsf{cntr} \rrbracket m+1}^{q-1} h(i) \leq \sum_{i=\llbracket \mathsf{cntr} \rrbracket m}^{q-1} h(i) \end{aligned}$$

Using the rules in Fig. 2, we can then extend this result to adversary calls and to the rest of the game, showing that $\llbracket G \rrbracket f \preceq f$.

Finally, let m be a memory such that $\neg F(m)$ and $[\![\mathsf{cntr}]\!]\ m = 0$. It follows immediately from $[\![G]\!] f \preceq f$ that

$$\Pr_{G,m}[F \wedge \mathsf{cntr} \leq q] \leq [\![G]\!]\ m\ f \leq f(m) = \sum_{i=0}^{q-1} h(i) \qquad \square$$

When failure is defined as the probability of a flag bad being set by an oracle and the number of queries the adversary makes to this oracle is upper bounded by q, the above lemma can be used to bound the probability of failure by taking $F = \mathsf{bad}$ and defining h suitably. In most practical applications (e.g. security of OAEP) h is a constant function; the proof of Lemma 1 given in Sec. 6.2 is an exception for which the full generality of the lemma is needed.

5 Eager and Lazy Sampling

Game-based proofs commonly include bridging steps in which one performs a semantics-preserving reordering of instructions. On most occasions, the reordering is intraprocedural. However, proofs in the random oracle model (see $\mathcal{O}_{\mathrm{lazy}}$ in Fig. 4 for an example of a random oracle) often use interprocedural code motion, in which sampling statements are moved from an oracle to the main command of the game or, conversely, from the main command to an oracle. The first transformation, called eager sampling, is useful for moving random choices upfront: a systematic application of eager sampling allows to transform a probabilistic game G that samples at most a fixed number of values into a semantically equivalent game $S; G'$, where S samples the values that might be needed in G, and G' is a completely deterministic program to the exception of adversaries that might still make their own random choices.[3] The second, dual, transformation, called lazy sampling, is useful to postpone sampling of random values until these values are actually used for the first time.

CertiCrypt features tactics that allow to perform and justify both intra and interprocedural code motion. The tactic for intraprocedural code motion is described in [3]. In this section, we present a general method to prove the correctness of interprocedural code motion. The method is based on a logic for swapping statements, and overcomes many limitations of our earlier lemma reported in [3]. A first limitation of our earlier lemma is that it only allowed to swap one random sampling at the time, whereas some applications, including the PRP/PRF Switching Lemma, require swapping a sequence of random samplings. Another limitation of our earlier method is that it could not be used for proving that some queries to a random oracle $\mathcal{O}$ are uniformly distributed and independent from the view of the adversary, as needed in the proof of IND-CCA of OAEP.

[3] Making adversaries deterministic is the goal of the *coin fixing* technique, as described in [5]; formalizing this technique is left for future work.

5.1 A Logic for Swapping Statements

The primary tool for performing eager/lazy sampling is an extension of the Relational Hoare Logic with rules for swapping statements. As the goal is to move code across procedures, it is essential that the logic considers two potentially different environments E and E'. The logic deals with judgments of the form

$$\vdash E, (c; S) \sim E', (S; c') : \Psi \Rightarrow \Phi$$

In most cases, the logic will be applied with S being a sequence of (guarded) sampling statements; however, the logic does not constrain S, and merely requires that S satisfies three basic properties:

$$\mathsf{modify}(E, S, X) \qquad \mathsf{modify}(E', S, X) \qquad \vdash E, S \simeq_X^{I \cup X} E', S$$

for some sets of variables X and I. Some rules of the logic are given in Fig. 3; for the sake of readability, all rules are specialized to $\equiv$, although we formalized more general versions of the rules, e.g. for conditional statements

$$\frac{\begin{array}{c}\vdash E, c_1; S \sim E', S; c_1') : P \wedge \langle e \rangle_1 \Rightarrow Q \qquad \vdash E, c_2; S \sim E', S; c_2' : P \wedge \langle \neg e \rangle_1 \Rightarrow Q \\ P \Rightarrow \langle e \rangle_1 = \langle e' \rangle_2 \qquad \vdash E', S \sim E', S : = \wedge \langle e' \rangle_1 \Rightarrow = \wedge \langle e' \rangle_1\end{array}}{\vdash E, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; S) \sim E', (S; \mathsf{if}\ e'\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2') : P \Rightarrow Q}$$

which is used in the application considered in the next section.

$$\frac{x \notin I \cup X \qquad \mathsf{fv}(e) \cap X = \emptyset}{\vdash E, (x \leftarrow e; S) \ \equiv\ E', (S; x \leftarrow e)} \qquad \frac{x \notin I \cup X}{\vdash E, (x \xleftarrow{\$} T; S) \ \equiv\ E', (S; x \xleftarrow{\$} T)}$$

$$\frac{\vdash E, (c_1; S) \ \equiv\ E', (S; c_1') \quad \vdash E, (c_2; S) \ \equiv\ E', (S; c_2')}{\vdash E, (c_1; c_2; S) \ \equiv\ E', (S; c_1'; c_2')}$$

$$\frac{\vdash E, (c_1; S) \ \equiv\ E', (S; c_1') \qquad \vdash E, (c_2; S) \ \equiv\ E', (S; c_2') \qquad \mathsf{fv}(e) \cup X = \emptyset}{\vdash E, (\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; S) \ \equiv\ E', (S; \mathsf{if}\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2')}$$

$$\frac{\vdash E, (c; S) \ \equiv\ E', (S; c') \quad \mathsf{fv}(e) \cup X = \emptyset}{\vdash E, (\mathsf{while}\ e\ \mathsf{do}\ c; S) \ \equiv\ E', (S; \mathsf{while}\ e\ \mathsf{do}\ c')}$$

$$\frac{\begin{array}{c}\vdash E, (f_E.\mathsf{body}; S) \ \equiv\ E', (S; f_{E'}.\mathsf{body}) \quad f_E.\mathsf{params} = f_{E'}.\mathsf{params} \quad f_E.\mathsf{re} = f_{E'}.\mathsf{re} \\ \mathsf{fv}(f_E.\mathsf{re}) \cap X = \emptyset \quad x \notin I \cup X \quad \mathsf{fv}(\boldsymbol{e}) \cap X = \emptyset\end{array}}{\begin{array}{c}\vdash E, (x \leftarrow f(\boldsymbol{e}); S) \ \equiv\ E', (S; x \leftarrow f(\boldsymbol{e})) \\ \vdash_{\mathrm{wf}} \mathcal{A} \quad X \cap (\mathcal{RW} \cup \mathcal{RO}) = \emptyset \quad I \cap \mathcal{RW} = \emptyset \quad \forall f \notin \mathcal{F}.\ f_E = f_{E'} \\ \forall f \in \mathcal{F}.\ f_E.\mathsf{params} = f_{E'}.\mathsf{params} \wedge f_E.\mathsf{re} = f_{E'}.\mathsf{re} \wedge \\ \vdash E, (f_E.\mathsf{body}; S) \ \equiv\ E', (S; f_{E'}.\mathsf{body}) \\ \hline \vdash E, (x \leftarrow \mathcal{A}(\boldsymbol{e}); S) \ \equiv\ E', (S; x \leftarrow \mathcal{A}(\boldsymbol{e}))\end{array}}$$

Fig. 3. Selected rules of a logic for swapping statements

5.2 Application

Consider the games $\mathsf{G}_{\mathsf{lazy}}$ and $\mathsf{G}_{\mathsf{eager}}$ in Fig. 4. Suppose that our goal is to provide an upper bound of the probability of an event A in game $\mathsf{G}_{\mathsf{lazy}}$ and that the proof proceeds by eagerly sampling the value that the oracle $\mathcal{O}$ returns in response for a particular query $\hat{\boldsymbol{x}}$. Define

$$S_{\hat{\boldsymbol{y}}} \stackrel{\text{def}}{=} \mathsf{if}\ \hat{\boldsymbol{x}} \notin \mathsf{dom}(\boldsymbol{L})\ \mathsf{then}\ \hat{\boldsymbol{y}} \xleftarrow{\$} T\ \mathsf{else}\ \hat{\boldsymbol{y}} \leftarrow \boldsymbol{L}(\hat{\boldsymbol{x}})$$

and take $I = \{\hat{\boldsymbol{x}}, \boldsymbol{L}\}$ and $X = \{\hat{\boldsymbol{y}}\}$. We have that

$$\vdash E_{\mathsf{lazy}},\ (c; S_{\hat{\boldsymbol{y}}}) \ \equiv\ E_{\mathsf{eager}},\ (S_{\hat{\boldsymbol{y}}}; c) \quad \Longrightarrow \quad \Pr_{\mathsf{G}_{\mathsf{lazy}}}[A] = \Pr_{\mathsf{G}_{\mathsf{eager}}}[A]$$

Game $\mathsf{G}_{\mathsf{lazy}}$:
$\boldsymbol{L} \leftarrow [\,];\ c$
Oracle $\mathcal{O}_{\mathsf{lazy}}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  $y \xleftarrow{\$} T$;
  $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$
else $y \leftarrow \boldsymbol{L}(x)$
return y

Game $\mathsf{G}_{\mathsf{eager}}$:
$\boldsymbol{L} \leftarrow [\,];\ \hat{\boldsymbol{y}} \xleftarrow{\$} T;\ c$
Oracle $\mathcal{O}_{\mathsf{eager}}(x)$:
if $x \notin \mathsf{dom}(\boldsymbol{L})$ then
  if $x = \hat{\boldsymbol{x}}$ then $y \leftarrow \hat{\boldsymbol{y}}$ else $y \xleftarrow{\$} T$;
  $\boldsymbol{L} \leftarrow (x, y) :: \boldsymbol{L}$
else $y \leftarrow \boldsymbol{L}(x)$
return y

Fig. 4. An example of eager sampling using interprocedural code motion

Thus, in order to move from game $\mathsf{G}_{\mathsf{lazy}}$ to game $\mathsf{G}_{\mathsf{eager}}$, it is enough to prove the commutation property on the left of the implication. This, in turn, requires showing that

$$\vdash E_{\mathsf{lazy}},\ (\mathcal{O}_{\mathsf{lazy}}.\mathsf{body}; S_{\hat{\boldsymbol{y}}}) \ \equiv\ E_{\mathsf{eager}},\ (S_{\hat{\boldsymbol{y}}}; \mathcal{O}_{\mathsf{eager}}.\mathsf{body})$$

which can be achieved by applying the rules of the logic for swapping statements and the relational Hoare logic.

6 Proofs of the PRP/PRF Switching Lemma

We have formalized two proofs of the Switching lemma: both use the Fundamental Lemma to bound the advantage of the adversary by the probability of a failure event. The first proof uses the eager sampling technique to bound the probability of failure, whereas the second one relies on the Failure Event Lemma.

We begin by introducing in Fig. 5 annotated versions $\mathsf{G}^{\mathsf{bad}}_{\mathsf{RP}}$ and $\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}}$ of the games G_{RP} and G_{RF} defined in Sec. 2. From Lemma 2, we readily have

$$\left|\Pr_{\mathsf{G}_{\mathsf{RP}}}[b = 1] - \Pr_{\mathsf{G}_{\mathsf{RF}}}[b = 1]\right| = \left|\Pr_{\mathsf{G}^{\mathsf{bad}}_{\mathsf{RP}}}[b = 1] - \Pr_{\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}}}[b = 1]\right| \leq \Pr_{\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}}}[\mathsf{bad}]$$

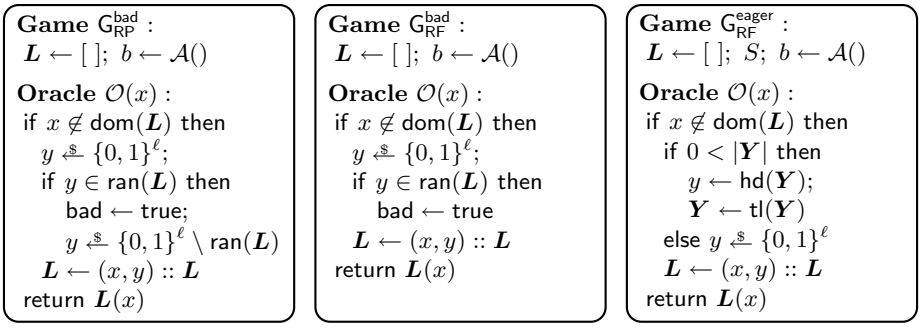


Fig. 5. Games used in the proofs of the PRP/PRF Switching Lemma

6.1 A Proof Based on Eager Sampling

We make a first remark: the probability of bad being set in game $\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}}$ is bounded by the probability of having a collision in $\mathsf{ran}(\boldsymbol{L})$ at the end of the game. Let us write this latter event as $\mathsf{col}(\boldsymbol{L})$. We prove this by showing that $\mathsf{bad} \implies \mathsf{col}(\boldsymbol{L})$ is an invariant of the game by means of the mechanized relational logic.

Using the logic for swapping statements, we modify the oracle in $\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}}$ so that the responses to the first q queries are instead chosen at the beginning of the game and stored in a list $\boldsymbol{Y}$, thus obtaining the equivalent eager version $\mathsf{G}^{\mathsf{eager}}_{\mathsf{RF}}$ shown in Fig. 5. Each time a query is made, the oracle pops a value from list $\boldsymbol{Y}$ and gives it back to the adversary as the response.

By using the rules of the logic for swapping statements, we show that the call $b \leftarrow \mathcal{A}()$ swaps with S. Since the initialization code S terminates and does not modify $\boldsymbol{L}$, we can conclude that

$$\Pr_{\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}}}[\mathsf{col}(\boldsymbol{L})] = \Pr_{\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}};S}[\mathsf{col}(\boldsymbol{L})] = \Pr_{\mathsf{G}^{\mathsf{eager}}_{\mathsf{RF}}}[\mathsf{col}(\boldsymbol{L})]$$

We prove using the Relational Hoare Logic that having a collision in the range of $\boldsymbol{L}$ at the end of this last game is the same as having a collision in $\boldsymbol{Y}$ immediately after executing S. We conclude that the bound in Eq. (1) holds by analyzing the loop in S. Observe that if there are no collisions in $\boldsymbol{Y}$ in a memory m, the probability of sampling a colliding value in the remaining loop iterations is

$$\Pr_{S,m}[\exists i, j \in \mathbb{N}, i < j < q \wedge \boldsymbol{Y}[i] = \boldsymbol{Y}[j]] = \sum_{i=|\boldsymbol{Y}|}^{q-1} \frac{i}{2^{\ell}}$$

This is proved by induction on $(q - |\boldsymbol{Y}|)$.

Remark. The proof reported in [3] uses a similar sequence of games. The sole difference is in the application of eager sampling. Here we do it in one step, whereas the earlier version uses induction. The new proof is both simpler and shorter (about 400 lines of Coq compared to the 900 lines of the former proof).

6.2 A Proof Based on Bounding Triples

The bound in Eq. 1 follows from a direct application of Lemma 3. It suffices to take $P = x \notin \mathsf{dom}(\boldsymbol{L})$, $F = \mathsf{bad}$, $h(i) = i\,2^{-\ell}$, and $\mathsf{cntr} = |\boldsymbol{L}|$. If bad is initially set to false in memory m, we have

$$\begin{aligned}\Pr_{\mathsf{G}^{\mathsf{bad}}_{\mathsf{RF}},m}[\mathsf{bad}] &= \Pr_{b\leftarrow\mathcal{A}(),m\{\boldsymbol{L}:=[]\}}[\mathsf{bad}] \\ &= \Pr_{b\leftarrow\mathcal{A}(),m\{\boldsymbol{L}:=[]\}}[\mathsf{bad} \wedge |\boldsymbol{L}| \le q] \le \sum_{i=0}^{q-1} h(i) = \frac{q(q-1)}{2^{\ell+1}}\end{aligned}$$

The second equation holds because $\mathcal{A}$ does not make more than q queries to oracle $\mathcal{O}$; the last inequality is obtained from Lemma 3. We use the logic in Fig. 2 to bound the probability of bad being set in one call to the oracle by $|\boldsymbol{L}|/2^{-\ell}$, as required by the Failure Event Lemma. The resulting proof is considerably shorter compared to the one presented in the previous section (only about 100 lines of Coq).

7 Related Work

We refer to [3] for an overview of papers that apply proof assistants to cryptography, and that focus on programming language techniques akin to those described in this paper. The first line of work is concerned with reasoning about probabilistic programs. Several program logics have been studied in the literature, see e.g. [7,12], and some authors have developed machine-checked frameworks to reason about randomized algorithms. Hurd et al [9] report on a formalization of the logic of [12], and on several applications. Aubebaud and Paulin [2] present another framework, which provides the library of probabilities upon which CertiCrypt is built. The second line of work is concerned with certified program transformations. Over the last few years, certified optimizing compilers have become a reality, see e.g. [11]. In the course of these efforts, many program transformations have been certified, including lazy code motion [16]. There is a similarity between lazy code sampling and rematerialization [6]—an optimization that recomputes a value instead of loading it from memory—and it would be interesting to see whether the method developed in this paper could prove useful to build a translation validator for the former.

8 Conclusion

The game-based approach to cryptographic proofs routinely uses a number of unusual programming language techniques. In this paper we report on the certification and automation of two such techniques, namely failure events, and eager/lazy sampling. Both techniques have been used extensively to successfully provide a machine-checked proof of IND-CCA security of the OAEP padding scheme. Our ultimate goal is to provide a comprehensive coverage of the techniques used by cryptographers, and to turn CertiCrypt into an effective platform for verifying a wide range of cryptographic proofs.

References

1. Affeldt, R., Tanaka, M., Marti, N.: Formal proof of provable security by game-playing in a proof assistant. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) ProvSec 2007. LNCS, vol. 4784, pp. 151–168. Springer, Heidelberg (2007)
2. Audebaud, P., Paulin-Mohring, C.: Proofs of randomized algorithms in Coq. Science of Computer Programming 74(8), 568–589 (2009)
3. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Proceedings of the 36th ACM Symposium on Principles of Programming Languages, pp. 90–101. ACM Press, New York (2009)
4. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Proceedings of the 1st ACM Conference on Computer and Communications Security, pp. 62–73. ACM Press, New York (1993)
5. 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)
6. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation, pp. 311–321. ACM Press, New York (1992)
7. Corin, R., den Hartog, J.: A probabilistic Hoare-style logic for game-based cryptographic proofs. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 252–263. Springer, Heidelberg (2006)
8. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. Journal of Cryptology 17(2), 81–104 (2004)
9. Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in HOL. Theor. Comput. Sci. 346(1), 96–112 (2005)
10. Impagliazzo, R., Rudich, S.: Limits on the provable consequences of one-way permutations. In: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, pp. 44–61. ACM Press, New York (1989)
11. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Proceedings of the 33rd ACM Symposium Principles of Programming Languages, pp. 42–54. ACM Press, New York (2006)
12. McIver, A., Morgan, C.: Abstraction, Refinement, and Proof for Probabilistic Systems. Springer, Heidelberg (2005)
13. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332 (2004), http://eprint.iacr.org/2004/332
14. Stern, J.: Why provable security matters? In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 449–461. Springer, Heidelberg (2003)
15. The Coq development team: The Coq Proof Assistant Reference Manual Version 8.2 (2009), http://coq.inria.fr
16. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 316–326. ACM Press, New York (2009)

Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder*

Jasmin Christian Blanchette and Tobias Nipkow

Institut für Informatik, Technische Universität München, Germany
{blanchette,nipkow}@in.tum.de

Abstract. Nitpick is a counterexample generator for Isabelle/HOL that builds on Kodkod, a SAT-based first-order relational model finder. Nitpick supports unbounded quantification, (co)inductive predicates and datatypes, and (co)recursive functions. Fundamentally a finite model finder, it approximates infinite types by finite subsets. As case studies, we consider a security type system and a hotel key card system. Our experimental results on Isabelle theories and the TPTP library indicate that Nitpick generates more counterexamples than other model finders for higher-order logic, without restrictions on the form of the formulas to falsify.

1 Introduction

Anecdotal evidence suggests that most "theorems" initially given to an interactive theorem prover do not hold, typically because of a typo or a missing assumption, but sometimes because of a fundamental flaw. Modern proof assistants often include counterexample generators that can be run on putative theorems or on specific subgoals in a proof to spare users the Sisyphean task of trying to prove non-theorems.

Isabelle/HOL [17] includes two such tools: Quickcheck [4] generates functional code for the higher-order logic (HOL) formula and evaluates it for random values of the free variables, and Refute [23] searches for finite countermodels of a formula through a reduction to SAT (Boolean satisfiability). Their areas of applicability are almost disjoint: Quickcheck excels at inductive datatypes but is restricted to the executable fragment of HOL (which excludes unbounded quantifiers) and may loop endlessly on inductive predicates. Refute copes well with logical symbols, but inductive datatypes and predicates are mostly out of reach due to combinatorial explosion.

In the first-order world, the Alloy Analyzer [13], a testing tool for first-order relational logic (FORL), has enjoyed considerable success lately. Alloy's backend, the relational model finder Kodkod [21], is available as a stand-alone Java library and is used in many projects.

Alloy's success inspired us to develop a new counterexample generator for Isabelle, called Nitpick.[1] It uses Kodkod as its backend, thereby benefiting from Kodkod's optimizations (notably its symmetry breaking) and its rich relational logic. The basic translation from HOL to FORL is conceptually simple (Section 3); however, common HOL

* This work is supported by the DFG grant Ni 491/11-1.

[1] The name Nitpick is appropriated from Alloy's venerable precursor.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 131–146, 2010.

idioms such as (co)inductive datatypes and (co)inductive predicates necessitate a translation scheme tailored for SAT solving (Section 4). In addition, Nitpick benefits from many novel optimizations that greatly improve its performance, especially in the presence of higher-order constructs (Section 5).

As case studies, we consider the Isabelle formalizations of a hotel key card system and a security type system (Section 6), both of which are currently beyond the reach of Quickcheck and Refute. Our evaluation indicates that Nitpick falsifies more formulas than Quickcheck and Refute (Section 7), to a large extent because it imposes no syntactic restrictions on the formulas to falsify. Nitpick is integrated with the TPTP benchmark suite [20] and exposed three bugs in the higher-order provers TPS [1] and LEO-II [3].

2 Background

2.1 Higher-Order Logic (HOL)

The types and terms of HOL [12] are that of the simply typed λ-calculus extended with type constructors and constants:

Types:		*Terms:*	
$\sigma ::= \alpha$	(type variable)	$t ::= x^\sigma$	(variable)
$\mid (\sigma, \ldots, \sigma)\,\kappa$	(type construction)	$\mid c^\sigma$	(constant)
		$\mid t\ t$	(application)
		$\mid \lambda x^\sigma.\ t$	(abstraction)

We write κ for $()\,\kappa$, $\sigma\,\kappa$ for $(\sigma)\,\kappa$, and $\sigma\,\kappa\,\tau$ for $(\sigma, \tau)\,\kappa$. HOL's standard semantics interprets the Boolean type o and the function space $\sigma \to \tau$. Other types are defined, notably the product type $\sigma \times \tau$. The function arrow associates to the right, reflecting the left-associativity of application. We assume throughout that terms are well-typed using the standard typing rules and write x and c instead of x^σ and c^σ when the type σ is irrelevant or can be inferred from the context. A formula is a term of type o.

Type variables occurring in the type of a constant can be instantiated, offering a restricted form of polymorphism. Standard models interpret the constant $\simeq^{\alpha\to\alpha\to o}$ as equality on α for any instance of α. Logical connectives and quantifiers can be defined in terms of $\simeq$; for example, $\mathit{True}^o = (\lambda x^o.\ x) \simeq (\lambda x.\ x)$ and $\forall^{(\alpha\to o)\to o} = (\lambda P^{\alpha\to o}.\ P \simeq (\lambda x.\ \mathit{True}))$. The traditional binder notation $Q x.\ t$ abbreviates $Q\ (\lambda x.\ t)$.

2.2 First-Order Relational Logic (FORL)

Kodkod's idiosyncratic logic, FORL, combines elements from first-order logic and relational calculus, to which it adds the transitive closure operator [21]. Its formulas involve variables and terms ranging over relations (sets of tuples drawn from a universe of uninterpreted atoms) of arbitrary arities. The logic is unsorted, but each term denotes a relation of a fixed arity that can be inferred from the arities of its variables. Our translation relies on the following FORL fragment.

Formulas:

$\varphi ::= \mathsf{false}$ (falsity)
$\mid \mathsf{true}$ (truth)
$\mid m\ r$ (multiplicity constraint)
$\mid r \simeq r$ (equality)
$\mid r \subseteq r$ (inclusion)
$\mid \neg\varphi$ (negation)
$\mid \varphi \wedge \varphi$ (conjunction)
$\mid \forall d\colon \varphi$ (universal quantification)

$d ::= x \in r$

$m ::= \mathsf{no} \mid \mathsf{lone} \mid \mathsf{one}$

$n ::= 1 \mid 2 \mid \cdots$

Terms:

$r ::= \mathsf{none}$ (empty set)
$\mid \mathsf{iden}$ (identity relation)
$\mid \mathsf{a}_n$ (atom)
$\mid x$ (variable)
$\mid \{\langle d,\ldots,d\rangle \mid \varphi\}$ (comprehension)
$\mid \pi_n^n(r)$ (projection)
$\mid r^+$ (transitive closure)
$\mid r.r$ (dot-join)
$\mid r \times r$ (Cartesian product)
$\mid r \cup r$ (union)
$\mid r - r$ (difference)
$\mid \mathsf{if}\ \varphi\ \mathsf{then}\ r\ \mathsf{else}\ r$ (conditional)

FORL syntactically distinguishes between terms and formulas. The universe of discourse is $\mathscr{A} = \{\mathsf{a}_1,\ldots,\mathsf{a}_k\}$, where each a_i is an uninterpreted atom. Atoms and n-tuples are identified with singleton sets and singleton n-ary relations, respectively. Bound variables in quantifications and comprehensions range over the tuples in a relation; thus, $\forall x \in (\mathsf{a}_1 \cup \mathsf{a}_2) \times \mathsf{a}_3\colon \varphi(x)$ is equivalent to $\varphi(\mathsf{a}_1 \times \mathsf{a}_3) \wedge \varphi(\mathsf{a}_2 \times \mathsf{a}_3)$.

Although they are not listed above, we will sometimes make use of $\vee$, $\longrightarrow$, $\exists$, *, and $\cap$ in examples. The constraint $\mathsf{no}\ r$ expresses that r is the empty relation, $\mathsf{one}\ r$ expresses that r is a singleton, and $\mathsf{lone}\ r \iff \mathsf{no}\ r \vee \mathsf{one}\ r$. The projection and dot-join operators are unconventional; their semantics is given by the equations

$$[\![\pi_i^k(r)]\!] = \{(r_i,\ldots,r_{i+k-1}) \mid (r_1,\ldots,r_m) \in [\![r]\!]\}$$
$$[\![r.s]\!] = \{(r_1,\ldots,r_{m-1},s_2,\ldots,s_n) \mid \exists t.\ (r_1,\ldots,r_{m-1},t) \in [\![r]\!] \wedge (t,s_2,\ldots,s_n) \in [\![s]\!]\}.$$

The dot-join operator admits three important special cases. Let s be unary and r, r' be binary relations. The expression $s.r$ gives the direct image of the set s under r; if s is a singleton and r a function, it coincides with the function application $r(s)$. Analogously, $r.s$ gives the inverse image of s under r. Finally, $r.r'$ expresses relational composition.

To pass an n-tuple s to a function r, we write $\langle s\rangle.r$, which stands for the n-fold dot-join $\pi_n(s).(\ldots.(\pi_1(s).r)\ldots)$. We write $\pi_i(r)$ for $\pi_i^1(r)$.

The relational operators often make it possible to express first-order problems concisely. The following Kodkod specification attempts to fit 30 pigeons in 29 holes:

vars $pigeons = \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\}$, $holes = \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}$
var $\emptyset \subseteq nest \subseteq \{\mathsf{a}_1,\ldots,\mathsf{a}_{30}\} \times \{\mathsf{a}_{31},\ldots,\mathsf{a}_{59}\}$
solve $(\forall p \in pigeons\colon \mathsf{one}\ p.nest) \wedge (\forall h \in holes\colon \mathsf{lone}\ nest.h)$

The example declares three free variables: *pigeons* and *holes* are given fixed values, whereas *nest* is specified with a lower and an upper bound. Variable declarations are an extralogical way of specifying sort constraints and partial solutions.

The constraint $\mathsf{one}\ p.nest$ states that pigeon p is in relation with exactly one hole, and $\mathsf{lone}\ nest.h$ that hole h is in relation with at most one pigeon. Taken as a whole, the formula states that *nest* is a one-to-one function. It is, of course, not satisfiable, a fact that Kodkod can establish in less than a second.

When reducing FORL to SAT, each n-ary relational variable y is in principle translated to an $|\mathscr{A}|^n$ array of propositional variables $V[i_1,\ldots,i_n]$, with $V[i_1,\ldots,i_n] \Longleftrightarrow \langle \mathsf{a}_{i_1},\ldots,\mathsf{a}_{i_n}\rangle \in y$. Most relational operations can be coded efficiently; for example, $\cup$ is simply $\vee$. The quantified formula $\forall r \in s.\, \varphi(r)$ is treated as $\bigwedge_{j=1}^{n} t_j \subseteq s \longrightarrow \varphi(t_j)$, where the t_j's are the tuples that may belong to s. Transitive closure is unrolled to saturation.

3 The Basic Translation

Nitpick employs Kodkod to find a finite model (a satisfying assignment to the free variables and constants) of $\neg P$, where P is the formula to refute. The translation of a formula from HOL to FORL is parameterized by the cardinalities of the types occurring in it, provided as a function $|\sigma|$ from types to positive integers obeying

$$|\sigma| \geq 1 \qquad |o| = 2 \qquad |\sigma \to \tau| = |\tau|^{|\sigma|} \qquad |\sigma \times \tau| = |\sigma| \cdot |\tau|\,.$$

Following Jackson [13], we call such a function a *scope*. Like other SAT-based model finders, Nitpick enumerates the possible scopes for each basic type, so that if a formula has a finite counterexample, the tool eventually finds it, unless it runs out of resources.

The basic translation presented in this section handles the following HOL constants:

False^{o}	(falsity)	$\mathit{insert}^{\alpha\to(\alpha\to o)\to\alpha\to o}$	(element insertion)
True^{o}	(truth)	$\mathit{UNIV}^{\alpha\to o}$	(universal set)
$\simeq^{\alpha\to\alpha\to o}$	(equality)	$\cup^{(\alpha\to o)\to(\alpha\to o)\to\alpha\to o}$	(union)
$\subseteq^{(\alpha\to o)\to(\alpha\to o)\to o}$	(subset)	$-^{(\alpha\to o)\to(\alpha\to o)\to\alpha\to o}$	(set difference)
$\neg^{o\to o}$	(negation)	$\mathit{Pair}^{\alpha\to\beta\to\alpha\times\beta}$	(pair constructor)
$\wedge^{o\to o\to o}$	(conjunction)	$\mathit{fst}^{\alpha\times\beta\to\alpha}$	(first projection)
$\forall^{(\alpha\to o)\to o}$	(universal quantifier)	$\mathit{snd}^{\alpha\times\beta\to\beta}$	(second projection)
$\emptyset^{\alpha\to o}$	(empty set)	$(\ )^{+\,(\alpha\times\alpha\to o)\to\alpha\times\alpha\to o}$	(transitive closure)

SAT solvers are particularly sensitive to the encoding of problems, so special care is needed when translating HOL formulas. Whenever practicable, HOL constants should be mapped to their FORL equivalents, rather than expanded to their definitions. This is especially true for the transitive closure r^+, which is defined as the least fixed point of $\lambda R\,(x,y).\,(\exists a\,b.\,x \simeq a \wedge y \simeq b \wedge r\,(a,b)) \vee (\exists a\,b\,c.\,x \simeq a \wedge y \simeq c \wedge R\,(a,b) \wedge r\,(b,c))$.

As a rule, HOL functions should be mapped to FORL relations accompanied by a constraint. For example, assuming the scope $|\alpha| = 2$ and $|\beta| = 3$, the presumptive theorem $\forall x^{\alpha}.\,\exists y^{\beta}.\,f\,x \simeq y$ corresponds to the Kodkod problem

var $\emptyset \subseteq f \subseteq \{\mathsf{a}_1,\mathsf{a}_2\} \times \{\mathsf{a}_3,\mathsf{a}_4,\mathsf{a}_5\}$
solve $(\forall x \in \mathsf{a}_1 \cup \mathsf{a}_2\colon \mathsf{one}\ x.f) \wedge \neg(\forall x \in \mathsf{a}_1 \cup \mathsf{a}_2\colon \exists y \in \mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5\colon x.f \simeq y)$

The first conjunct ensures that f is a function, and the second conjunct is the negation of the HOL formula translated to FORL.

An n-ary first-order function (curried or not) can be coded as an $(n+1)$-ary relation accompanied by a constraint. However, if the return type is o, the function is more efficiently coded as an unconstrained n-ary relation. This allows formulas such as $A^+ \cup B^+ \simeq (A \cup B)^+$ to be translated without taking a detour through ternary relations.

Higher-order quantification and functions bring complications of their own. For example, we would like to translate $\forall g^{\beta\to\alpha}.\ g\,x \not\simeq y$ into something like

$$\forall g \subseteq (\mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5) \times (\mathsf{a}_1 \cup \mathsf{a}_2)\colon (\forall x \in \mathsf{a}_3 \cup \mathsf{a}_4 \cup \mathsf{a}_5\colon \mathsf{one}\ x.g) \longrightarrow x.g \not\simeq y,$$

but the $\subseteq$ symbol is not allowed at the binding site; only $\in$ is. Skolemization solves half of the problem (Section 5.1), but for the remaining quantifiers we are forced to adopt an unwieldy n-tuple singleton representation of functions, where n is the cardinality of the domain. For the formula above, this gives

$$\forall G \in (\mathsf{a}_1 \cup \mathsf{a}_2)\times(\mathsf{a}_1 \cup \mathsf{a}_2)\times(\mathsf{a}_1 \cup \mathsf{a}_2)\colon x.(\overbrace{\mathsf{a}_3\times\pi_1(G)\ \cup\ \mathsf{a}_4\times\pi_2(G)\ \cup\ \mathsf{a}_5\times\pi_3(G)}^{g}) \not\simeq y,$$

where G is the triple corresponding to g. In the body, we convert the singleton G to the relational representation, then we apply x on it using dot-join. The singleton encoding is also used for passing functions to functions; fortunately, two optimizations, function specialization and boxing (Section 5.1), make this rarely necessary.

We are now ready to look at the basic translation in more detail. The translation distinguishes between formulas (F), singletons (S), and relations (R). We start by mapping HOL types to sets of FORL atom tuples. For each type σ, we provide two codings, a singleton representation $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ and a relational representation $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$:[2]

$$\begin{aligned}
\mathsf{S}\langle\!\langle\sigma\to\tau\rangle\!\rangle &= \mathsf{S}\langle\!\langle\tau\rangle\!\rangle^{|\sigma|} & \qquad \mathsf{R}\langle\!\langle\sigma\to o\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \\
\mathsf{S}\langle\!\langle\sigma\times\tau\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{S}\langle\!\langle\tau\rangle\!\rangle & \qquad \mathsf{R}\langle\!\langle\sigma\to\tau\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{R}\langle\!\langle\tau\rangle\!\rangle \\
\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle &= \{\mathsf{a}_1,\ldots,\mathsf{a}_{|\sigma|}\} & \qquad \mathsf{R}\langle\!\langle\sigma\rangle\!\rangle &= \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle.
\end{aligned}$$

In the S representation, an element of type σ is mapped to a single tuple $\in \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$. In the R representation, an element of type $\sigma\to o$ is mapped to a subset of $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ consisting of the points at which the predicate is *True*; an element of $\sigma\to\tau$ (where $\tau \neq o$) is mapped to a relation $\subseteq \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle \times \mathsf{R}\langle\!\langle\tau\rangle\!\rangle$; any other element is coded as a singleton. For simplicity, we reuse the same atoms for distinct types. Doing so is sound for well-typed terms.

For each free variable y^σ, we generate the declaration $\mathbf{var}\ \emptyset \subseteq y \subseteq \mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$ as well as a constraint $\Phi^\sigma(y)$ to ensure that functions are functions and single values are singletons:

$$\Phi^{\sigma_1\to\cdots\to\sigma_n\to o}(r) = \mathsf{true} \qquad \Phi^{\sigma\to\tau}(r) = \forall b_\mathrm{f} \in \mathsf{S}\langle\!\langle\sigma\rangle\!\rangle\colon \Phi^\tau\left(\langle b_\mathrm{f}\rangle.r\right) \qquad \Phi^\sigma(r) = \mathsf{one}\ r.$$

We assume that free and bound variables are syntactically distinguishable, and use the letter y for the former and b for the latter. The symbol b_f denotes a fresh bound variable.

We assume a total order on n-tuples of atoms and let $\mathsf{S}_i\langle\!\langle\sigma\rangle\!\rangle$ denote the ith tuple from $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ according to that order. Furthermore, we define $s(\sigma)$ and $r(\sigma)$ as the arity of the tuples in $\mathsf{S}\langle\!\langle\sigma\rangle\!\rangle$ and $\mathsf{R}\langle\!\langle\sigma\rangle\!\rangle$, respectively. The translation of terms requires the following rather technical conversions between singletons (S), relations (R), and formulas (F):

$$\begin{aligned}
s2r^{\sigma\to o}(r) &= \textstyle\bigcup_{i=1}^{|\sigma|} \pi_i(r).(\mathsf{a}_2 \times \mathsf{S}_i\langle\!\langle\sigma\rangle\!\rangle) & \qquad f2s(\varphi) &= \mathsf{if}\ \varphi\ \mathsf{then}\ \mathsf{a}_2\ \mathsf{else}\ \mathsf{a}_1 \\
s2r^{\sigma\to\tau}(r) &= \textstyle\bigcup_{i=1}^{|\sigma|} \mathsf{S}_i\langle\!\langle\sigma\rangle\!\rangle \times s2r^\tau\left(\pi^{s(\tau)}_{(i-1)\cdot s(\tau)+1}(r)\right) & \qquad s2f(r) &= r \simeq \mathsf{a}_2 \\
r2s^{\sigma\to\tau}(r) &= \{\langle b_\mathrm{f} \in \mathsf{S}\langle\!\langle\sigma\to\tau\rangle\!\rangle\rangle \mid s2r^{\sigma\to\tau}(b_\mathrm{f}) \simeq r\} & \qquad s2r^\sigma(r) &= r \qquad r2s^\sigma(r) = r.
\end{aligned}$$

The Boolean values false and true are arbitrarily coded as a_1 and a_2, respectively.

[2] Metatheoretic functions here and elsewhere are defined using sequential pattern matching, eliminating the need for side conditions such as "if $\tau \neq o$" and "otherwise."

The translation of HOL terms is performed by three functions, $\mathsf{F}\langle\!\langle t \rangle\!\rangle$, $\mathsf{S}\langle\!\langle t \rangle\!\rangle$, and $\mathsf{R}\langle\!\langle t \rangle\!\rangle$. Their defining equations are to be matched modulo η-equivalence:

$$\begin{array}{rcl}
\mathsf{F}\langle\!\langle y \rangle\!\rangle = s2f(y) & \mathsf{F}\langle\!\langle t \subseteq u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle u \rangle\!\rangle & \mathsf{S}\langle\!\langle b \rangle\!\rangle = b \\
\mathsf{F}\langle\!\langle b \rangle\!\rangle = s2f(b) & \mathsf{F}\langle\!\langle \neg t \rangle\!\rangle = \neg \mathsf{F}\langle\!\langle t \rangle\!\rangle & \mathsf{S}\langle\!\langle \mathit{Pair}\ t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle t \rangle\!\rangle \times \mathsf{S}\langle\!\langle u \rangle\!\rangle \\
\mathsf{F}\langle\!\langle \mathit{False} \rangle\!\rangle = \mathsf{false} & \mathsf{F}\langle\!\langle t \wedge u \rangle\!\rangle = \mathsf{F}\langle\!\langle t \rangle\!\rangle \wedge \mathsf{F}\langle\!\langle u \rangle\!\rangle & \mathsf{S}\langle\!\langle \mathit{fst}\ t^{\sigma\times\tau} \rangle\!\rangle = \pi_1^{s(\sigma)}(\mathsf{S}\langle\!\langle t \rangle\!\rangle) \\
\mathsf{F}\langle\!\langle \mathit{True} \rangle\!\rangle = \mathsf{true} & \mathsf{F}\langle\!\langle \forall b^{\sigma}.\ t \rangle\!\rangle = \forall b \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle\colon \mathsf{F}\langle\!\langle t \rangle\!\rangle & \mathsf{S}\langle\!\langle \mathit{snd}\ t^{\sigma\times\tau} \rangle\!\rangle = \pi^{s(\tau)}_{s(\sigma)+1}(\mathsf{S}\langle\!\langle t \rangle\!\rangle) \\
\mathsf{F}\langle\!\langle t \simeq u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle \simeq \mathsf{R}\langle\!\langle u \rangle\!\rangle & \mathsf{F}\langle\!\langle t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle u \rangle\!\rangle \subseteq \mathsf{R}\langle\!\langle t \rangle\!\rangle & \mathsf{S}\langle\!\langle t \rangle\!\rangle = r2s^{\sigma}\big(\mathsf{R}\langle\!\langle t \rangle\!\rangle\big)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{R}\langle\!\langle c^{o} \rangle\!\rangle = f2s(c) & \mathsf{R}\langle\!\langle \mathit{insert}\ t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle t \rangle\!\rangle \cup \mathsf{R}\langle\!\langle u \rangle\!\rangle \\
\mathsf{R}\langle\!\langle y \rangle\!\rangle = y & \mathsf{R}\langle\!\langle t \cup u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle \cup \mathsf{R}\langle\!\langle u \rangle\!\rangle \\
\mathsf{R}\langle\!\langle b^{\sigma} \rangle\!\rangle = s2r^{\sigma}(b) & \mathsf{R}\langle\!\langle t - u \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle - \mathsf{R}\langle\!\langle u \rangle\!\rangle \\
\mathsf{R}\langle\!\langle \mathit{Pair}\ t\ u \rangle\!\rangle = \mathsf{S}\langle\!\langle \mathit{Pair}\ t\ u \rangle\!\rangle & \mathsf{R}\langle\!\langle (t^{\sigma\times\sigma\to o})^{+} \rangle\!\rangle = \mathsf{R}\langle\!\langle t \rangle\!\rangle^{+} \quad \text{if } r(\sigma) = 1 \\
\mathsf{R}\langle\!\langle \mathit{fst}\ t^{\sigma\times\tau} \rangle\!\rangle = s2r^{\sigma}\big(\mathsf{S}\langle\!\langle \mathit{fst}\ t \rangle\!\rangle\big) & \mathsf{R}\langle\!\langle t^{\sigma\to o}\ u \rangle\!\rangle = f2s\big(\mathsf{F}\langle\!\langle t\ u \rangle\!\rangle\big) \\
\mathsf{R}\langle\!\langle \mathit{snd}\ t^{\sigma\times\tau} \rangle\!\rangle = s2r^{\tau}\big(\mathsf{S}\langle\!\langle \mathit{snd}\ t \rangle\!\rangle\big) & \mathsf{R}\langle\!\langle t\ u \rangle\!\rangle = \langle \mathsf{S}\langle\!\langle u \rangle\!\rangle \rangle . \mathsf{R}\langle\!\langle t \rangle\!\rangle \\
\mathsf{R}\langle\!\langle \emptyset^{\sigma} \rangle\!\rangle = \mathsf{none}^{r(\sigma)} & \mathsf{R}\langle\!\langle \lambda b^{\sigma}.\ t^{o} \rangle\!\rangle = \{\langle b \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle \rangle \mid \mathsf{F}\langle\!\langle t \rangle\!\rangle\} \\
\mathsf{R}\langle\!\langle \mathit{UNIV}^{\sigma} \rangle\!\rangle = \mathsf{R}\langle\!\langle \sigma \rangle\!\rangle & \mathsf{R}\langle\!\langle \lambda b^{\sigma}.\ t^{\tau} \rangle\!\rangle = \{\langle b \in \mathsf{S}\langle\!\langle \sigma \rangle\!\rangle, b_{\mathrm{f}} \in \mathsf{R}\langle\!\langle \tau \rangle\!\rangle \rangle \mid b_{\mathrm{f}} \subseteq \mathsf{R}\langle\!\langle t \rangle\!\rangle\}.
\end{array}$$

Annoyingly, the translation of transitive closure is defined only if $r(\sigma) = 1$. We will see ways to lift this restriction in the next two sections.

Theorem 1 (Soundness). *Given a putative theorem P with free variables $y_1^{\sigma_1}, \ldots, y_n^{\sigma_n}$ within our HOL fragment and a scope S, P admits a counterexample if there exists a valuation V with $V(y_j) \subseteq \mathsf{R}\langle\!\langle \sigma_j \rangle\!\rangle$ that satisfies the FORL formula $\mathsf{F}\langle\!\langle \neg P \rangle\!\rangle \wedge \bigwedge_{j=1}^{n} \Phi^{\sigma_j}(y_j)$.*

Proof sketch. Let $[\![t]\!]_A$ denote the set-theoretic semantics of the HOL term t w.r.t. a variable assignment A and the scope S. Let $[\![\rho]\!]_V$ denote the semantics of the FORL term or formula ρ w.r.t. a variable valuation V and the scope S. Furthermore, let $\lfloor v \rfloor_{\mathsf{X}}$ denote the X-encoded FORL value corresponding to the HOL value v, for $\mathsf{X} \in \{\mathsf{F}, \mathsf{S}, \mathsf{R}\}$. Using recursion induction, it is straightforward to prove that $[\![\mathsf{X}\langle\!\langle t \rangle\!\rangle]\!]_V = \lfloor [\![t]\!]_A \rfloor_{\mathsf{X}}$ if $V(y_i) = \lfloor A(y_i) \rfloor_{\mathsf{R}}$ for all free variables y_i and $V(b_i) = \lfloor A(b_i) \rfloor_{\mathsf{S}}$ for all locally free bound variables b_i occurring in t. Moreover, from the satisfying valuation V of the free variables y_i, we can construct a type-correct HOL assignment A such that $\lfloor A(y_i) \rfloor_{\mathsf{R}} = V(y_i)$; the $\Phi^{\sigma_j}(y_j)$ constraints and the variable bounds $V(y_j) \subseteq \mathsf{R}\langle\!\langle \sigma_j \rangle\!\rangle$ ensure that such an assignment exists. Hence, $[\![\mathsf{F}\langle\!\langle \neg P \rangle\!\rangle]\!]_V = \mathsf{true} = \lfloor [\![\neg P]\!]_A \rfloor_{\mathsf{F}}$, which shows that A falsifies P.

A very thorough soundness proof of a translation from HOL to SAT can be found in Tjark Weber's Ph.D. thesis [23].

4 Refinements to the Basic Translation

4.1 Approximation of Infinite Types and Partiality

Because of the axiom of infinity, the type *nat* of natural numbers does not admit any finite models. To work around this, Nitpick considers finite subsets $\{0, 1, \ldots, K-1\}$

of *nat* and maps numbers $\geq K$ to the undefined value ($\bot$), coded as none, the empty set. Formulas of the form $\forall n^{nat}.\ P(n)$ are treated in essence as $(\forall n < K.\ P(n)) \wedge P(\bot)$, which usually evaluates to either *False* (if $P(i)$ gives *False* for some $i < K$) or $\bot$, but not to *True*, since we do not know whether $P(K)$, $P(K+1)$, ... (collectively represented by $P(\bot)$) are true. In view of this, Nitpick generally cannot soundly disprove conjectures that contain an infinite existential quantifier in their conclusion or an infinite universal quantifier in their assumptions. As a fallback, the tool enters an unsound mode in which the quantifiers are artificially bounded. Counterexamples obtained under these conditions are marked as "potential."

Functions from *nat* to α are abstracted by relations $\subseteq \{\mathsf{a}_1, \ldots, \mathsf{a}_K\} \times \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\}$ constrained to be partial functions. Partiality makes it possible to encode the successor function *Suc* as the relation $S = (\mathsf{a}_1 \times \mathsf{a}_2) \cup \cdots \cup (\mathsf{a}_{K-1} \times \mathsf{a}_K)$, which associates no value with a_K. Conveniently, the dot-join $\mathsf{a}_K.S$ yields none, and so does $\mathsf{none}.S$. This is desirable because *Suc* $(K-1)$ is unrepresentable and *Suc* $\bot$ is unknown.

Partiality leads to a Kleene three-valued logic, which is expressed in terms of Kodkod's two-valued logic as follows. At the outermost level, we let the FORL truth value false stand for both *False* (no counterexample) and $\bot$ (potential counterexample), and reserve true for *True* (genuine counterexample). The same convention is obeyed in other positive contexts within the formula. In negative contexts, false codes *False* and true codes *True* or $\bot$. Finally, in unpolarized contexts (for example, as argument to a function), the atom a_1 codes *False*, a_2 codes *True*, and none codes $\bot$. Unlike similar approximation approaches [18, p. 164; 23], Nitpick's logic is sound, although the tool also has an unsound mode as noted above.

4.2 Nonuniform Representation of HOL Terms

FORL gives Nitpick a lot of flexibility when encoding terms. A value of type $\alpha \times \beta$, for example, can be translated as before to a pair $\in \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha|}\} \times \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\beta|}\}$, but it can also be mapped to a single atom $\in \{\mathsf{a}_1, \ldots, \mathsf{a}_{|\alpha \times \beta|}\}$. Predicates on α (or functions from α to σ with $|\sigma| = 2$) can be coded as single atoms, sets of atoms, relations from atoms to $\{\mathsf{a}_1, \mathsf{a}_2\}$, or $|\alpha|$-tuples over $\{\mathsf{a}_1, \mathsf{a}_2\}$.

Nitpick uses FORL's flexibility to a larger extent than was hinted at in Section 3. For example, it ensures that the operand of transitive closure is always a binary relation (a set of pairs), no matter what the HOL type is, lifting an annoying limitation in the basic translation described earlier. It also keeps track of whether a term can evaluate to $\bot$, which makes many optimizations possible in FORL. For example, because free variables never yield $\bot$, we encode $x \simeq y$ as $x \subseteq y$, which is more efficient.

The current representation selection scheme proceeds in a straightforward bottom-up fashion, inserting conversions as appropriate. More sophisticated schemes that would minimize the number of conversions have yet to be tried.

4.3 Encoding of (Co)Inductive Predicates

Isabelle lets users specify (co)inductive predicates p by their introduction rules and synthesizes a fixed point definition $p \simeq \mathit{lfp}\ F$ or $p \simeq \mathit{gfp}\ F$. For performance reasons,

Nitpick avoids expanding *lfp* and *gfp* to their definitions and translates (co)inductive predicates directly, using appropriate FORL concepts.

A first intuition is that an inductive predicate p is a fixed point, so we could use the equation $p \simeq F\ p$ as the axiomatic specification of p. In general, this is unsound since it underspecifies p, but there are two important cases for which this method is sound. First, if the recursion in F is well-founded, the fixed point equation $p \simeq F\ p$ admits exactly one solution and we can safely use it as p's specification. Second, if p occurs negatively in the formula, we can replace these occurrences by a fresh constant q satisfying the axiom $q \simeq F\ q$; this transformation preserves equisatisfiability.

To deal with positive occurrences of p, we adapt a technique from bounded model checking [5]: We replace p by a fresh predicate r_k defined by

$$r_0 \simeq (\lambda \bar{x}.\ \bot) \qquad\qquad r_{n+1} \simeq F\ r_n,$$

which corresponds to p unrolled k times. For unpolarized occurrences, we use $q \cap r_k$. In essence, we have made p well-founded by adding a counter that decreases by one with each recursive call. This unrolling comes at a price: The search space and the size of the propositional formula for r_k is k times that of q. Hence, it makes sense to look for a counterexample with a small value of k first and increment it gradually if needed.

The situation is mirrored for coinductive predicates: Negative occurrences of p become r_k, positive occurrences become q, and unpolarized occurrences become $q \cup r_k$.

To determine whether a predicate is well-founded, Nitpick generates a wellfoundedness goal and invokes Isabelle's termination prover [7] with a time limit. Given introduction rules of the form

$$\frac{p\ \bar{t}_{i1} \quad \cdots \quad p\ \bar{t}_{in_i} \quad Q_i}{p\ \bar{u}_i}$$

for $i \in \{1,\ldots,m\}$, the termination prover must exhibit a well-founded relation R such that $\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n_i} Q_i \longrightarrow \langle \bar{t}_{ij}, \bar{u}_i \rangle \in R$ holds.

In our experience, about half of the inductive predicates occurring in practice are well-founded—this includes most type systems and other compositional formalisms, but generally excludes state transition systems.

4.4 Encoding of (Co)Inductive Datatypes and (Co)Recursive Functions

In contrast to Isabelle's constructor-oriented treatment of inductive datatypes, Nitpick's FORL axiomatization revolves around selectors and discriminators, inspired by Kuncak and Jackson's modeling of lists and trees in Alloy [14]. The selector and discriminator view is usually more efficient than the constructor view because it breaks high-arity constructors into several low-arity selectors.

Consider the type $\alpha\ list$ generated from $Nil^{\alpha\ list}$ and $Cons^{\alpha \to \alpha\ list \to \alpha\ list}$. The FORL axiomatization is done in terms of the discriminators $isNil^{\alpha\ list \to o}$ and $isCons^{\alpha\ list \to o}$ and the selectors $get1Cons^{\alpha\ list \to \alpha}$ and $get2Cons^{\alpha\ list \to \alpha\ list}$, which give access to a nonempty list's head and tail. Following Dunets et al. [10], *Nil* and *Cons x xs* are translated as $isNil$ and $get1Cons{\cdot}x \cap get2Cons{\cdot}xs$, respectively.

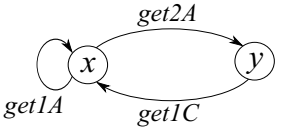
The following axioms, with $N = 1, 2$, specify a subterm-closed finite universe of lists using the atoms $\mathscr{A}_{\alpha\ list}$:

$$\begin{aligned}
&\textsc{Disj}: && \mathsf{no}\ \mathit{isNil} \cap \mathit{isCons}\\
&\textsc{Exh}: && \mathit{isNil} \cup \mathit{isCons} \simeq \mathscr{A}_{\alpha\ \mathit{list}}\\
&\textsc{Sel}_N: && \forall \mathit{xs} \in \mathscr{A}_{\alpha\ \mathit{list}}.\ \mathsf{if}\ \mathit{xs} \subseteq \mathit{isCons}\ \mathsf{then\ one}\ \mathit{xs}.\mathit{getNCons}\ \mathsf{else\ no}\ \mathit{xs}.\mathit{getNCons}\\
&\textsc{Uniq}: && \mathsf{lone}\ \mathit{isNil} \wedge (\forall x \in \mathscr{A}_{\alpha}, \mathit{xs} \in \mathscr{A}_{\alpha\ \mathit{list}}.\ \mathsf{lone}\ \mathit{get1Cons}.x \cap \mathit{get2Cons}.\mathit{xs})\\
&\textsc{Acycl}: && \mathsf{no}\ \mathit{get2Cons}^{+} \cap \mathsf{iden}.
\end{aligned}$$

Examples of subterm-closed list universes using traditional list notation are $\{[], [\mathsf{a}_1], [\mathsf{a}_2], [\mathsf{a}_3]\}$ and $\{[], [\mathsf{a}_2], [\mathsf{a}_3, \mathsf{a}_2], [\mathsf{a}_1, \mathsf{a}_3, \mathsf{a}_2]\}$. For recursive functions, Nitpick ignores the construction synthesized by Isabelle and relies instead on the user-specified equations.

The approach can be generalized to mutually recursive datatypes. To generate the Acyclic axioms for the mutually recursive datatypes x with constructors $A^{x \to y \to x}$ and B^{x} and y with constructor $C^{x \to y}$, we compute their datatype dependency graph, in which vertices are labeled with datatypes and arcs are labeled with selectors. Then we compute for each datatype a regular expression capturing the nontrivial paths from the datatype to itself, with $\cdot$ standing for concatenation, $\cup$ for alternative, and * and $^+$ for repetition. We require the paths to be disjoint from identity:

$$\begin{aligned}
&\mathsf{no}\ (\mathit{get1A} \cup \mathit{get2A}.\mathit{get1C})^{+} \cap \mathsf{iden}\\
&\mathsf{no}\ (\mathit{get1C}.\mathit{get1A}^{*}.\mathit{get2A})^{+} \cap \mathsf{iden}.
\end{aligned}$$



Nitpick supports coinductive datatypes, even though Isabelle does not provide a high-level mechanism for defining them. Users can define custom coinductive datatypes from first principles and tell Nitpick to substitute its efficient FORL axiomatization for their definitions. Nitpick also knows about Isabelle's coinductive "lazy list" datatype, $\alpha\ \mathit{llist}$, with the constructors $\mathit{LNil}^{\alpha\ \mathit{llist}}$ and $\mathit{LCons}^{\alpha \to \alpha\ \mathit{llist} \to \alpha\ \mathit{llist}}$. The FORL axiomatization is similar to that used for $\alpha\ \mathit{list}$, but the Acyclic axiom is omitted to allow cyclic (ω-regular) lists. Infinite lists are presented to the user as lassos, with a finite stem and cycle. The following coinductive bisimulation principle is translated along with the HOL formula, to ensure that distinct atoms correspond to observably distinct lists:

$$\frac{}{\mathit{LNil} \simeq \mathit{LNil}}\ \textsc{Bisim}_1 \qquad \frac{x \simeq y \quad \mathit{xs} \simeq \mathit{ys}}{\mathit{LCons}\ x\ \mathit{xs} \simeq \mathit{LCons}\ y\ \mathit{ys}}\ \textsc{Bisim}_2.$$

5 Optimization Steps

5.1 HOL Preprocessing

Function Specialization. A function argument is said to be static if it is passed unaltered to all recursive calls. A typical example is f in the definition of *map*:

$$\mathit{map}\ f\ [] \simeq [] \qquad\qquad \mathit{map}\ f\ (x \cdot \mathit{xs}) \simeq f\ x \cdot \mathit{map}\ f\ \mathit{xs}.$$

An optimization reminiscent of the static argument transformation or lambda-dropping [9, pp. 148–156] is to specialize the function for each eligible call site, thereby avoiding passing the static argument altogether. At the call site, any term whose free variables

are all globally free is eligible for this optimization. Following this scheme, *map Suc ns* would become $map_{Suc}\ ns$, where map_{Suc} is defined as follows:

$$map_{Suc}\ [] \simeq [] \qquad\qquad map_{Suc}\ (x \cdot xs) \simeq Suc\ x \cdot map_{Suc}\ xs.$$

For this example, specialization reduces the number of propositional variables needed to encode the function by a factor of $|nat|^{|nat|}$.

Boxing. Nitpick normally translates function and product types directly to the homologous Kodkod concepts. This is not always desirable; for example, a transition relation on states represented as n-tuples leads to a $2n$-ary relation, which gives rise to a combinatorial explosion and precludes the use of FORL's binary transitive closure.

Our experience suggests that it is almost always advantageous to approximate n-tuples where $n \geq 3$ as well as higher-order arguments. This is achieved by wrapping them in an isomorphic type $\alpha\ box$ with the single constructor $Box^{\alpha \to \alpha\ box}$, inserting constructors and selectors as appropriate. Assuming that specialization is not in use, the second equation for *map* would then become

$$map\ f^{(nat \to nat)\ box}\ (x \cdot xs) \simeq get1Box\ f\ x \cdot map\ f\ xs,$$

with *map* (*Box Suc*) *ns* at the call site. Notice that for function types, boxing is similar to defunctionalization [2], with selectors playing the role of "apply" functions. Further opportunities for boxing are created by uncurrying high-arity constants beforehand.

Quantifier Massaging. (Co)inductive definitions are marred by existential quantifiers, which blow up the size of the resulting propositional formula. The following steps are applied to eliminate quantifiers or reduce their binding range: (1) Replace quantifications of the forms $\forall x.\ x \simeq t \longrightarrow P(x)$ and $\exists x.\ x \simeq t \wedge P(x)$ by $P(t)$ if x does not occur free in t. (2) Skolemize. (3) Distribute quantifiers over congenial connectives ($\forall$ over $\wedge$, $\exists$ over $\vee$ and $\longrightarrow$). (4) For any remaining subformula $Qx_1 \ldots x_n.\ p_1 \otimes \cdots \otimes p_m$, where Q is a quantifier and $\otimes$ is a connective, move the p_i's out of as many quantifiers as possible by rebuilding the formula using $qfy(\{x_1,\ldots,x_n\}, \{p_1,\ldots,p_m\})$, defined as

$$qfy(\emptyset, P) = \bigotimes P \qquad\qquad qfy(x \uplus X, P) = qfy(X, P - P_x \cup \{Qx.\ \bigotimes P_x\}),$$

where $P_x = \{p \in P \mid x \text{ occurs free in } p\}$.

The order in which individual variables x are removed from the first argument is crucial because it affects which p_i's can be moved out. For clusters of up to 7 quantifiers, Nitpick considers all permutations of the bound variables and chooses the one that minimizes the sum $\sum_{i=1}^{m} |\tau_{i1}| \cdot \ldots \cdot |\tau_{ik_i}| \cdot size(p_i)$, where $\tau_{i1},\ldots,\tau_{ik_i}$ are the types of the variables that have p_i in their binding range, and $size(p_i)$ is a rough syntactic measure of p_i's size; for larger clusters, it falls back on a heuristic inspired by Paradox's clause splitting procedure [8]. Thus, the formula $\exists x^{\alpha}\ y^{\alpha}.\ p\ x \wedge q\ x\ y \wedge r\ y\ (f\ y\ y)$ is transformed into $\exists y^{\alpha}.\ r\ y\ (f\ y\ y) \wedge (\exists x^{\alpha}.\ p\ x \wedge q\ x\ y)$. Processing y before x in *qfy* would instead give $\exists x^{\alpha}.\ p\ x \wedge (\exists y^{\alpha}.\ q\ x\ y \wedge r\ y\ (f\ y\ y))$, which is more expensive because $r\ y\ (f\ y\ y)$, the most complex conjunct, is doubly quantified and hence $|\alpha|^2$ copies of it are needed in the resulting propositional formula.

Constructor Elimination. Since datatype constructors may return $\perp$ in our encoding, we canincrease precision by eliminating them. A formula such as $[x,y] \simeq [a,b]$ can

easily be rewritten into $x \simeq a \wedge y \simeq b$, which evaluates to either *True* or *False* even if $[x,y]$ or $[a,b]$ would yield $\bot$.

For multiple-argument constructors, eliminating constructors helps reduce the number of nested quantifiers. Consider a datatype of AVL trees with two constructors, $Null^{\alpha\ tree}$ and $Node^{\alpha\to\alpha\ tree\to\alpha\ tree\to nat\to\alpha\ tree}$, and a *data* constant defined by the equations

$$data\ Null \simeq \emptyset \qquad \forall a\ t_1\ t_2\ h.\ data\ (Node\ a\ t_1\ t_2\ h) \simeq \{a\} \cup data\ t_1 \cup data\ t_2.$$

Our target is the constructor application $Node\ a\ t_1\ t_2\ h$ in the second equation's left-hand side. We first pull it out and assign it to a fresh bound variable y:

$$\forall a\ t_1\ t_2\ h\ y.\ \ y \simeq Node\ a\ t_1\ t_2\ h \longrightarrow data\ y \simeq \{a\} \cup data\ t_1 \cup data\ t_2.$$

Then we express the constructor arguments in terms of selectors in the conclusion, rewrite the assumption to use a discriminator, and omit the obsolete variables:

$$\forall y.\ isNode\ y \longrightarrow data\ y \simeq \{get1Node\ y\} \cup data\ (get2Node\ y) \cup data\ (get3Node\ y).$$

By quantifying over a single variable, we reduce the number of copies of the body from $|\alpha| \cdot |\alpha\ tree|^2 \cdot |nat|$ to $|\alpha\ tree|$ in the SAT problem, without losing counterexamples. This technique is also useful for constructors taking a single higher-order argument, such as those inserted by the boxing optimization described above.

5.2 Monotonicity Inference

Many formulas occurring in practice are monotonic in the sense that if the formula is falsifiable for a given scope, it is also falsifiable for all larger scopes [13, p. 165]. That not all formulas are monotonic will become clear after considering $|UNIV| = 3$.

Monotonicity can be exploited to prune the search space. For a formula involving n uninterpreted types, a model finder must a priori consider k^n scopes to exhaust all models up to the cardinality bound k. With monotonicity, it is sufficient to consider the single scope in which all types have cardinality k.

We developed and implemented two calculi for inferring monotonicity, and proved them sound [6]. The first calculus, on which we focus here, has limited support for sets encoded as predicates. The second, more powerful calculus addresses this problem by annotating function arrows and relying on a SAT solver to ensure consistent annotations.

For simplicity of exposition, the first calculus is defined for a HOL fragment in which the only constants are $\simeq$ and $\longrightarrow$. We let *True* abbreviate $(\lambda x^o.\ x) \simeq (\lambda x.\ x)$ and $\forall x.\ p$ abbreviate $(\lambda x.\ p) \simeq (\lambda x.\ True)$. We assume a distinguished type variable α with respect to which monotonicity is inferred. The calculus is defined below:

$$\mathbf{TV}^s(o) = \emptyset \qquad \mathbf{TV}^+(\beta) = \{\beta\} \qquad \mathbf{TV}^-(\beta) = \emptyset \qquad \mathbf{TV}^s(\sigma \to \tau) = \mathbf{TV}^{-s}(\sigma) \cup \mathbf{TV}^s(\tau)$$

$$\frac{}{\mathbf{K}(x)} \qquad \frac{}{\mathbf{K}(\longrightarrow)} \qquad \frac{\alpha \notin \mathbf{TV}^-(\sigma)}{\mathbf{K}(\simeq^{\sigma\to\sigma\to o})} \qquad \frac{\mathbf{K}(t) \quad \mathbf{K}(u)}{\mathbf{K}(t\ u)} \qquad \frac{\mathbf{K}(t)}{\mathbf{K}(\lambda x.\ t)}$$

$$\frac{\mathbf{K}(t)}{\mathbf{M}^s(t)} \qquad \frac{\mathbf{M}^{-s}(t) \quad \mathbf{M}^s(u)}{\mathbf{M}^s(t \longrightarrow u)} \qquad \frac{\mathbf{M}^+(t) \quad \alpha \notin \mathbf{TV}^+(\sigma)}{\mathbf{M}^+(\forall x^\sigma.\ t)} \qquad \frac{\mathbf{M}^-(t)}{\mathbf{M}^-(\forall x.\ t)} \qquad \frac{\mathbf{K}(t) \quad \mathbf{K}(u)}{\mathbf{M}^-(t \simeq u)}.$$

The $\mathbf{TV}^s(\sigma)$ function gives the set of type variables occurring positively (if s is $+$) or negatively (if s is $-$) with respect to $\to$. The judgment $\mathbf{K}(t)$ expresses that t's value remains essentially the same when α's cardinality is increased, assuming that the free variables also stay the same. A formula P is monotonic if $\mathbf{M}^{-}(P)$ is derivable.

We evaluated both calculi on the theorems from six highly polymorphic Isabelle theories (*AVL2*, *Fun*, *Huffman*, *List*, *Map*, and *Relation*). We found that the simple calculus inferred monotonicity for 41% to 97% of the theorems depending on the theory, while the more sophisticated calculus achieved 65% to 100% [6].

6 Case Studies

6.1 Volpano–Smith–Irvine Security Type System

Assuming a partition of program variables into public and private ones, Volpano, Smith, and Irvine [22] provide typing rules guaranteeing that the contents of private variables stay private. They define two types, *High* (private) and *Low* (public). An expression is *High* if it involves private variables; otherwise it is *Low*. A command is *High* if it modifies private variables only; commands that could alter public variables are *Low*.

As our first case study, we consider a fragment of the formal soundness proof by Snelting and Wasserrab [19]. Given a variable partition Γ, the inductive predicate $\Gamma \vdash e : \sigma$ tells whether e has type σ, whereas $\Gamma, \sigma \vdash c$ tells whether command c has type σ. Below is a flawed definition of $\Gamma, \sigma \vdash c$:

$$\frac{}{\Gamma, \sigma \vdash \texttt{skip}} \qquad \frac{\Gamma\, v \simeq \lfloor \mathit{High} \rfloor}{\Gamma, \sigma \vdash v := e} \qquad \frac{\Gamma \vdash e : \mathit{Low} \quad \Gamma\, v \simeq \lfloor \mathit{Low} \rfloor}{\Gamma, \mathit{Low} \vdash v := e} \qquad \frac{\Gamma, \sigma \vdash c_1}{\Gamma, \sigma \vdash c_1\,;\, c_2}$$

$$\frac{\Gamma \vdash b : \sigma \quad \Gamma, \sigma \vdash c_1 \quad \Gamma, \sigma \vdash c_2}{\Gamma, \sigma \vdash \texttt{if (}b\texttt{) } c_1 \texttt{ else } c_2} \qquad \frac{\Gamma \vdash b : \sigma \quad \Gamma, \sigma \vdash c}{\Gamma, \sigma \vdash \texttt{while (}b\texttt{) } c} \qquad \frac{\Gamma, \mathit{High} \vdash c}{\Gamma, \mathit{Low} \vdash c}.$$

The following theorem constitutes a key step in the soundness proof:

$$\Gamma, \mathit{High} \vdash c \;\wedge\; \langle c, s\rangle \rightsquigarrow^* \langle \texttt{skip}, s'\rangle \longrightarrow \forall v.\, \Gamma\, v \simeq \lfloor \mathit{Low} \rfloor \longrightarrow s\, v \simeq s'\, v.$$

Informally, it asserts that if executing the *High* command c in state s terminates in state s', then the public variables of s and s' must agree. This is consistent with our intuition that *High* commands should only modify private variables. However, because we planted a bug in the definition of $\Gamma, \sigma \vdash c$, Nitpick finds a counterexample:

$$\Gamma = [\mathtt{v}_1 \mapsto \mathit{Low}] \qquad s = [\mathtt{v}_1 \mapsto \mathit{false}]$$
$$c = \texttt{skip ; } \mathtt{v}_1 \texttt{ := } (\mathit{Var}\, \mathtt{v}_1 \texttt{ == } \mathit{Var}\, \mathtt{v}_1) \qquad s' = [\mathtt{v}_1 \mapsto \mathit{true}].$$

Even though the command c has type *High*, it assigns *true* to the *Low* variable $\mathtt{v}_1$. The bug is a missing assumption $\Gamma, \sigma \vdash c_2$ in the typing rule for sequential composition.

6.2 Hotel Key Card System

We consider a state-based model of a vulnerable hotel key card system with recordable locks [16], inspired by an Alloy specification due to Jackson [13, pp. 299–306]. The formalization relies on three opaque types, *room*, *guest*, and *key*. A key card, of

type $card = key \times key$, combines an old key and a new key. A state is a 7-field record $(\!|\mathsf{owns} :: room \to guest\ option, \mathsf{curr} :: room \to key, \mathsf{issued} :: key \to o, \mathsf{cards} :: guest \to card \to o, \mathsf{roomk} :: room \to key, \mathsf{isin} :: room \to guest \to o, \mathsf{safe} :: room \to o|\!)$. The set *reach* of reachable states is defined inductively by the following rules:

$$\frac{}{(\!|\mathsf{owns} = (\lambda r.\ \bot), \mathsf{curr} = init, \mathsf{issued} = range\ init, \mathsf{cards} = (\lambda g.\ \emptyset),\ \mathsf{roomk} = init, \mathsf{isin} = (\lambda r.\ \emptyset), \mathsf{safe} = (\lambda r.\ True)|\!) \in reach}\ \textsc{Init}$$

$$\frac{s \in reach \quad k \notin issued\ s}{s(\!|\mathsf{curr} := (curr\ s)(r := k), \mathsf{issued} := issued\ s \cup k,\ \mathsf{cards} := (cards\ s)(g := cards\ s\ g \cup \langle curr\ s\ r, k\rangle),\ \mathsf{owns} := (owns\ s)(r := \lfloor g \rfloor), \mathsf{safe} := (safe\ s)(r := False)|\!) \in reach}\ \textsc{Check-In}$$

$$\frac{s \in reach \quad \langle k, k'\rangle \in cards\ s\ g \quad roomk\ s\ r \in \{k, k'\}}{s(\!|\mathsf{isin} := (isin\ s)(r := isin\ s\ r \cup g), \mathsf{roomk} := (roomk\ s)(r := k'),\ \mathsf{safe} := (safe\ s)(r := owns\ s\ r \simeq \lfloor g \rfloor \wedge isin\ s\ r \simeq \emptyset \vee safe\ s\ r)|\!) \in reach}\ \textsc{Entry}$$

$$\frac{s \in reach \quad g \in isin\ s\ r}{s(\!|\mathsf{isin} := (isin\ s)(r := isin\ s\ r - \{g\})|\!) \in reach}\ \textsc{Exit}.$$

A desirable property of the system is that it should prevent unauthorized access:

$$s \in reach \wedge safe\ s\ r \wedge g \in isin\ s\ r \longrightarrow owns\ s\ r \simeq \lfloor g \rfloor.$$

Nitpick needs some help to contain the state space explosion: We restrict the search to one room and two guests. Within seconds, we get the counterexample

$$\begin{aligned} s = (\!|&\mathsf{owns} = (\mathsf{r}_1 := \lfloor \mathsf{g}_1 \rfloor), \mathsf{curr} = (\mathsf{r}_1 := \mathsf{k}_1), \mathsf{issued} = \{\mathsf{k}_1, \mathsf{k}_2, \mathsf{k}_3, \mathsf{k}_4\},\\ &\mathsf{cards} = (\mathsf{g}_1 := \{\langle \mathsf{k}_3, \mathsf{k}_1\rangle, \langle \mathsf{k}_4, \mathsf{k}_2\rangle\}, \mathsf{g}_2 := \{\langle \mathsf{k}_2, \mathsf{k}_3\rangle\}), \mathsf{roomk} = (\mathsf{r}_1 := \mathsf{k}_3),\\ &\mathsf{isin} = (\mathsf{r}_1 := \{\mathsf{g}_1, \mathsf{g}_2\}), \mathsf{safe} = \{\mathsf{r}_1\}|\!) \end{aligned}$$

with $g = \mathsf{g}_2$ and $r = \mathsf{r}_1$.

To retrace the steps from the initial state to the s, we can ask Nitpick to show the interpretation of *reach* at each iteration. This reveals the following "guest in the middle" attack: (1) Guest g_1 checks in and gets a card $\langle \mathsf{k}_4, \mathsf{k}_2\rangle$ for room r_1, whose lock expects k_4. Guest g_1 does not enter the room yet. (2) Guest g_2 checks in, gets a card $\langle \mathsf{k}_2, \mathsf{k}_3\rangle$ for r_1, and waits. (3) Guest g_1 checks in again, gets a card $\langle \mathsf{k}_3, \mathsf{k}_1\rangle$, inadvertently unlocks room r_1 with her previous card, $\langle \mathsf{k}_4, \mathsf{k}_2\rangle$, leaves a diamond on the nightstand, and exits. (4) Guest g_2 enters the room and "borrows" the diamond.

This flaw was already detected by Jackson using the Alloy Analyzer on his original specification and can be fixed by adding $k' \simeq curr\ s\ r$ to the conjunction in ENTRY.

7 Evaluation

An ideal way to assess Nitpick's strength would be to run it against Refute and Quickcheck on a representative database of Isabelle/HOL non-theorems. Lacking such a database, we chose instead to derive formulas from existing theorems by mutation, replacing constants with other constants and swapping arguments, as was done when

evaluating Quickcheck [4]. The vast majority of formulas obtained this way are invalid, and those few that are valid do not influence the ranking of the counterexample generators. For executable theorems, we made sure that the generated mutants are also executable to prevent a bias against Quickcheck.

The table below summarizes the results of running the tools on 3200 random mutants from 20 Isabelle theories (200 per theory), with a limit of 10 seconds per formula. Most counterexamples are found within a few seconds; giving the tool more time would have little impact on the results.

Theory	Quick.	Ref.	Nitp.	Theory	Quick.	Ref.	Nitp.
Divides	$134_{/184}$	3_{+15}	$\mathbf{141}_{+2}$	*ArrowGS*	$0_{/0}$	0_{+126}	$\mathbf{139}_{+2}$
Fun	$5_{/9}$	162_{+1}	$\mathbf{163}_{+0}$	*Coinductive*	$4_{/6}$	16_{+7}	$\mathbf{87}_{+13}$
GCD	$119_{/162}$	1_{+17}	$\mathbf{124}_{+10}$	*CoreC++*	$7_{/30}$	3_{+6}	$\mathbf{29}_{+1}$
List	$78_{/130}$	3_{+113}	$\mathbf{117}_{+9}$	*FFT*	$31_{/40}$	1_{+2}	$\mathbf{47}_{+15}$
MacLaurin	$\mathbf{43}_{/62}$	0_{+0}	26_{+7}	*Huffman*	$84_{/160}$	1_{+45}	$\mathbf{119}_{+2}$
Map	$19_{/34}$	103_{+45}	$\mathbf{157}_{+0}$	*MiniML*	$14_{/33}$	0_{+116}	$\mathbf{79}_{+49}$
Predicate	$2_{/2}$	147_{+14}	$\mathbf{161}_{+0}$	*NBE*	$41_{/62}$	0_{+18}	$\mathbf{81}_{+24}$
Relation	$0_{/2}$	144_{+3}	$\mathbf{150}_{+1}$	*Ordinal*	$0_{/66}$	10_{+3}	$\mathbf{12}_{+0}$
Set	$17_{/25}$	149_{+0}	$\mathbf{151}_{+0}$	*POPLmark*	$56_{/96}$	4_{+6}	$\mathbf{103}_{+15}$
Wellfounded	$10_{/24}$	118_{+20}	$\mathbf{141}_{+1}$	*Topology*	$0_{/0}$	124_{+4}	$\mathbf{139}_{+3}$

The table's entries have the form $G_{/X}$ for Quickcheck and G_{+P} for Refute and Nitpick, where $G =$ number of genuine counterexamples found and reported as such, $P =$ number of potential counterexamples found (in addition to G), and $X =$ number of executable mutants (among 200).

Refute's three-valued logic is unsound, so all counterexamples for formulas that involve an infinite type are potentially spurious and reported as such to the user. Nitpick also has an unsound mode, which contributes some potential counterexamples. Unfortunately, there is no easy way to tell how many of these are actually genuine.

Quickcheck and Nitpick are comparable on executable formulas, but Nitpick also fares well on non-executable ones. Notable exceptions are formalizations involving real arithmetic (*MacLaurin* and *FFT*), complex set-theoretic constructions (*Ordinal*), or a large state space (*CoreC++*).

Independently, Nitpick competes against Refute in the higher-order model finding division of the TPTP [20]. In a preliminary run, it disproved 293 out of 2729 formulas (mostly theorems), compared with 214 for Refute. Much to our surprise, Nitpick exhibited counterexamples for five formulas that had previously been proved by TPS [1] or LEO-II [3], revealing two bugs in the former and one bug in the latter.

8 Related Work

The approaches for testing conjectures can be classified in three broad categories:

– *Random testing.* The formula is evaluated for random values of the free variables. This approach is embodied by Isabelle's Quickcheck [4] and similar tools for other proof assistants. It is restricted to executable formulas.

– *SAT solving.* The formula is translated to propositional logic and handed to a SAT solver. This procedure was pioneered by McCune in his first-order finder MACE [15]. Other first-order MACE-style finders include Paradox [8] and Kodkod [21]. The higher-order finders Refute [23] and Nitpick also belong to this category.

– *Direct search.* The search for a model is performed directly on the formula, without translation to propositional logic. This approach was introduced by SEM [24].

Some proof methods deliver sound or unsound counterexamples upon failure, notably model checking, semantic tableaux, and satisfiability modulo theory (SMT) solving. Also worth of mention is the Dynamite tool [11], which lets users prove Alloy formulas in the interactive theorem prover PVS. Weber [23, pp. 3–4] provides a more detailed discussion of related work.

9 Conclusion

Nitpick is to our knowledge the first higher-order model finder that supports both inductive and coinductive predicates and datatypes. It works by translating higher-order formulas to first-order relational logic (FORL) and invoking the highly-optimized SAT-based Kodkod model finder [21] to solve these. Compared with Quickcheck, which is restricted to executable formulas, Nitpick shines by its generality—the hallmark of SAT-based model finding.

The translation to FORL is designed to exploit Kodkod's strengths. Datatypes are encoded following an Alloy idiom [10, 14] extended to mutually recursive and coinductive datatypes. FORL's relational operators provide a natural encoding of partial application and λ-abstraction, and the transitive closure plays a crucial role in the encoding of inductive datatypes. Our main contributions have been to isolate three ways to translate (co)inductive predicates to FORL, based on wellfoundedness, polarity, and linearity, and to devise optimizations—notably function specialization, boxing, and monotonicity inference—that dramatically increase scalability in practical applications.

Nitpick is included with the latest version of Isabelle and is invoked automatically whenever users enter new formulas to prove, helping to catch errors early, thereby saving time and effort. But Nitpick's real beauty is that it lets users experiment with formal specifications in the playful way championed by Alloy but with Isabelle's higher-order syntax, definition principles, and theories at their fingertips.

Acknowledgment. Stefan Berghofer, Lukas Bulwahn, Marcelo Frias, Florian Haftmann, Alexander Krauss, Mark Summerfield, Emina Torlak, and several anonymous reviewers provided useful comments on drafts of this paper. Alexander Krauss also helped devise the monotonicity inference calculi, and Geoff Sutcliffe ran Nitpick against Refute on the TPTP benchmark suite. We thank them all.

References

1. Andrews, P.B., Bishop, M., Issar, S., Nesmith, D., Pfenning, F., Xi, H.: TPS: A theorem-proving system for classical type theory. J. Auto. Reas. 16(3), 321–353 (1996)
2. Bell, J.M., Bellegarde, F., Hook, J.: Type-driven defunctionalization. ACM SIGPLAN Notices 32(8), 25–37 (1997)

3. Benzmüller, C., Paulson, L., Theiss, F., Fietzke, A.: Progress report on LEO-II, an automatic theorem prover for higher-order logic. In: Schneider, K., Brandt, J. (eds.) TPHOLs: Emerging Trends. C.S. Dept., University of Kaiserslautern, Internal Report 364/07 (2007)
4. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) SEFM 2004, pp. 230–239. IEEE C.S., Los Alamitos (2004)
5. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
6. Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, Springer, Heidelberg (to appear, 2010)
7. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in Isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer, Heidelberg (2007)
8. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: MODEL (2003)
9. de Medeiros Santos, A.L.: Compilation by Transformation in Non-Strict Functional Languages. Ph.D. thesis, C.S. Dept., University of Glasgow (1995)
10. Dunets, A., Schellhorn, G., Reif, W.: Bounded relational analysis of free datatypes. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 99–115. Springer, Heidelberg (2008)
11. Frias, M.F., Pombo, C.G.L., Moscato, M.M.: Alloy Analyzer+PVS in the analysis and verification of Alloy specifications. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 587–601. Springer, Heidelberg (2007)
12. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
14. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H.C. (ed.) ESEC/FSE 2005 (2005)
15. McCune, W.: A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, ANL (1994)
16. 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)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Schumann, J.M.: Automated Theorem Proving in Software Engineering. Springer, Heidelberg (2001)
19. Snelting, G., Wasserrab, D.: A correctness proof for the Volpano/Smith security typing system. In: Klein, G., Nipkow, T., Paulson, L.C. (eds.) AFP (September 2008)
20. Sutcliffe, G., Suttner, C.: The TPTP problem library for automated theorem proving, http://www.cs.miami.edu/~tptp/
21. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
22. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Comp. Sec. 4(3), 167–187 (1996)
23. Weber, T.: SAT-Based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Dept. of Informatics, T.U. München (2008)
24. Zhang, J., Zhang, H.: SEM: A system for enumerating models. In: Kaufmann, M. (ed.) IJCAI 95, vol. 1, pp. 298–303 (1995)

Formal Proof of a Wave Equation Resolution Scheme: The Method Error*

Sylvie Boldo[1,2], François Clément[3], Jean-Christophe Filliâtre[2,1], Micaela Mayero[4,5], Guillaume Melquiond[1,2], and Pierre Weis[3]

[1] INRIA Saclay – Île-de-France, ProVal, Orsay, F-91893
[2] LRI, Université Paris-Sud, CNRS, Orsay, F-91405
[3] INRIA Paris – Rocquencourt, Estime, Le Chesnay, F-78153
[4] LIPN, Université Paris 13, LCR, Villetaneuse, F-93430
[5] LIP, Arénaire (INRIA Grenoble - Rhône-Alpes, CNRS UMR 5668, UCBL, ENS Lyon), Lyon, F-69364

Abstract. Popular finite difference numerical schemes for the resolution of the one-dimensional acoustic wave equation are well-known to be convergent. We present a comprehensive formalization of the simplest scheme and formally prove its convergence in Coq. The main difficulties lie in the proper definition of asymptotic behaviors and the implicit way they are handled in the mathematical pen-and-paper proofs. To our knowledge, this is the first time this kind of mathematical proof is machine-checked.

Keywords: partial differential equation, acoustic wave equation, numerical scheme, Coq formal proofs.

1 Introduction

Ordinary differential equations (ODE) and partial differential equations (PDE) are ubiquitous in engineering and scientific computing. They show up in weather forecast, nuclear simulation, etc., and more generally in numerical simulation. Solutions to nontrivial problems are nonanalytic, hence approximated by numerical schemes over discrete grids.

Numerical analysis is mainly interested in proving the convergence of these schemes, that is, the approximation quality increases as the size of the discretization steps decreases. The approximation quality is characterized by the error defined as the difference between the exact continuous solution and the approximated discrete solution; this error must tend toward zero in order for the numerical scheme to be useful.

There is a wide literature on this topic, *e.g.* see [1,2], but no article goes into all the details. These "details" may have been skipped for readability, but they could also be mandatory details that were omitted due to an oversight. The

* This research was supported by the ANR projects CerPAN (ANR-05-BLAN-0281-04) and F∮ST (ANR-08-BLAN-0246-01).

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 147–162, 2010.

purpose of a mechanically-checked proof is to uncover these issues and check whether they could jeopardize the correctness of the schemes.

This work is a first step toward the development of formal tools for dealing with the convergence of numerical schemes. It would have been sensible to start with classical schemes for ODE, such as the Euler or Runge-Kutta methods. But we decided to directly validate the feasibility of our approach on the more complicated PDE. Moreover, this opens the door to a wide variety of applications, as they appear in many realistic problems from industry.

We chose the domain of wave propagation because it represents one of the most common physical phenomena one experiences in everyday life: directly through sight and hearing, but also via telecommunications, radar, medical imaging, etc. Industrial applications range from aeroacoustics to music acoustics (acoustic waves), from oil prospection to nondestructive testing (elastic waves), from optics to stealth technology (electromagnetic waves), and even include stabilization of ships and offshore platforms (surface gravity waves). We restrained ourselves to the simplest example of wave propagation models, the acoustic wave equation in a one-dimensional space domain, for it is a prototype model for all other kinds of wave. In this case, the equation describes the propagation of pressure variations (or sound waves) in a fluid medium; it also models the behavior of a vibrating string. For simplicity, we only consider homogeneous media, meaning that the propagation velocity is constant. Among the wide variety of numerical schemes for approximately solving the 1D acoustic wave equation, we chose the simplest one: the second order centered finite difference scheme, also known as the "three-point scheme". Again, for simplicity, we only consider regular grids with constant discretization steps for time and space.

To our knowledge, this is the first time this kind of mathematical proof is machine-checked.[1] Few works have been done on formalization and proofs on mathematical analysis inside proof assistants, and fewer on numerical analysis. Even real analysis developments are relatively new. The first developments on real numbers and real analysis are from the late 90's [3,4,5,6,7]. Some intuitionist formalizations have been realized by a team at Nijmegen [8,9]. Analysis results are available in provers such as ACL2, Coq, HOL Light, Isabelle, Mizar, or PVS. Regarding numerical analysis, we can cite [10] which deals, more precisely, with the formal proof of an automatic differentiation algorithm. About $\mathbb{R}^n$ and the dot product, an extensive work has been done by Harrison [11]. About the big O operator for asymptotic comparison, a decision procedure has been developed in [12]; unfortunately, we needed a more powerful big O and those results were not applicable.

Section 2 presents the PDE, the numerical scheme, and their mathematical properties. Section 3 describes the basic blocks of the formalization: dot product, big O, and Taylor expansions. Section 4 is devoted to the formal proof of the convergence of the numerical scheme.

[1] The Coq sources of the formal development are available from `http://fost.saclay.inria.fr/wave_method_error.php`

2 Wave Equation

A partial differential equation modeling an evolutionary problem is an equation involving partial derivatives of an unknown function of several independent space and time variables. The uniqueness of the solution is obtained by imposing additional conditions, typically the value of the function and the value of some of its derivatives at the initial time. The right-hand sides of such initial conditions are also called *Cauchy data*, making the whole problem a *Cauchy problem*, or an *initial-value problem*.

The mathematical theory is simpler when unbounded domains are considered [1]. When the space domain is bounded, the computation is simpler, but we have to take reflections at domain boundaries into account; this models a finite vibrating string fixed at both ends. Thanks to the nice property of finite velocity of propagation of the wave equation, we can build two Cauchy problems, one bounded and the other one unbounded, that coincide on the domain of the bounded one. Thus, we can benefit from the best of both worlds: the bounded problem makes computation simpler and the unbounded one avoids handling reflections. This section, as well as the steps taken at section 4 to conduct the proof of the convergence of the numerical scheme, is inspired by [13].

2.1 The Continuous Equation

The chosen PDE models the propagation of waves along an ideal vibrating elastic string, see [14,15]. It is obtained from Newton's laws of motion [16].

The gravity is neglected, hence the string is supposed rectilinear when at rest. Let $u(x,t)$ be the transverse displacement of the point of the string of abscissa x at time t from its equilibrium position. It is a (signed) scalar. Let c be the constant propagation velocity. It is a positive number that depends on the section and density of the string. Let $s(x,t)$ be the external action on the point of abscissa x at time t; it is a source term, such that $t = 0 \Rightarrow s(x,t) = 0$. Finally, let $u_0(x)$ and $u_1(x)$ be the initial position and velocity of the point of abscissa x. We consider the Cauchy problem (*i.e.*, with conditions at $t = 0$)

$$\forall t \geq 0,\ \forall x \in \mathbb{R}, \qquad (L(c)\,u)(x,t) \stackrel{\text{def}}{=} \frac{\partial^2 u}{\partial t^2}(x,t) + A(c)\,u(x,t) = s(x,t), \tag{1}$$

$$\forall x \in \mathbb{R}, \qquad (L_1\,u)(x,0) \stackrel{\text{def}}{=} \frac{\partial u}{\partial t}(x,0) = u_1(x), \tag{2}$$

$$\forall x \in \mathbb{R}, \qquad (L_0\,u)(x,0) \stackrel{\text{def}}{=} u(x,0) = u_0(x) \tag{3}$$

where the differential operator $A(c)$ is defined by

$$A(c) \stackrel{\text{def}}{=} -c^2 \frac{\partial^2}{\partial x^2}. \tag{4}$$

We admit that under reasonable conditions on the Cauchy data u_0 and u_1 and on the source term s, there exists a unique solution to the Cauchy problem (1)–(3) for each $c > 0$. This is a mathematical known fact (established for example from d'Alembert's formula (6)), that is left unproved here.

For such a solution u, it is natural to associate at each time t the positive definite quadratic quantity

$$E(c)(u)(t) \stackrel{\text{def}}{=} \frac{1}{2}\left\| x \mapsto \frac{\partial u}{\partial t}(x,t)\right\|^2 + \frac{1}{2}\left\| x \mapsto u(x,t)\right\|^2_{A(c)} \tag{5}$$

where $\langle v, w\rangle \stackrel{\text{def}}{=} \int_{\mathbb{R}} v(x)w(x)dx$, $\|v\|^2 \stackrel{\text{def}}{=} \langle v, v\rangle$ and $\|v\|^2_{A(c)} \stackrel{\text{def}}{=} \langle A(c)\, v, v\rangle$. The first term is interpreted as the kinetic energy, and the second term as the potential energy, making E the mechanical energy of the vibrating string.

This simple partial derivative equation happens to possess an analytical solution given by the so-called d'Alembert's formula [17], obtained from the method of characteristics [18], $\forall t \geq 0$, $\forall x \in \mathbb{R}$,

$$u(x,t) = \frac{1}{2}(u_0(x-ct) + u_0(x+ct)) + \frac{1}{2c}\int_{x-ct}^{x+ct} u_1(y)dy + \\ \frac{1}{2c}\int_0^t \left(\int_{x-c(t-\sigma)}^{x+c(t-\sigma)} s(y,\sigma)dy\right) d\sigma. \tag{6}$$

One can deduce from formula (6) the useful property of finite velocity of propagation. Assuming that we are only interested in the resolution of the Cauchy problem on a compact time interval of the form $[0, t_{\max}]$ with $t_{\max} > 0$, we suppose that u_0, u_1 and s have a compact support. Then the property states that there exists $x_{\min}$ and $x_{\max}$ with $x_{\min} < x_{\max}$ such that the support of the solution is a subset of $\Omega \stackrel{\text{def}}{=} [x_{\min}, x_{\max}] \times [0, t_{\max}]$. Furthermore, since the boundaries do not have time to be reached by the signal, the Cauchy problem set on Ω by adding homogeneous Dirichlet boundary conditions (*i.e.* for all $t \in [0, t_{\max}]$, $u(x_{\min}, t) = u(x_{\max}, t) = 0$), admits the same solution. Hence, we will numerically solve the Cauchy problem on Ω, but with the assumption that the spatial boundaries are not reached.

Note that the implementation of the compact spatial domain $[x_{\min}, x_{\max}]$ will be abstracted by the notion of finite support (that is to say, being zero outside of an interval, see Section 4.2) and will not appear explicitly otherwise.

Note also that most properties of the continuous problem proved unnecessary in the formalization of the numerical scheme and the proof of its convergence. For instance, integration operators and d'Alembert's formula can be avoided as long as we suppose the existence and regularity of a solution to the PDE and that this solution has a finite support.

2.2 The Discrete Equations

Let $(\Delta x, \Delta t)$ be a point in the interior of Ω; define the discretization functions $j_{\Delta x}(x) \stackrel{\text{def}}{=} \left\lfloor \frac{x - x_{\min}}{\Delta x} \right\rfloor$ and $k_{\Delta t}(t) \stackrel{\text{def}}{=} \left\lfloor \frac{t}{\Delta t} \right\rfloor$; then set $j_{\max} \stackrel{\text{def}}{=} j_{\Delta x}(x_{\max})$ and $k_{\max} \stackrel{\text{def}}{=} k_{\Delta t}(t_{\max})$. Now, the compact domain Ω is approximated by the regular discrete grid defined by

$$\forall k \in [0..k_{\max}],\ \forall j \in [0..j_{\max}], \quad \mathbf{x}_j^k \stackrel{\text{def}}{=} (x_j, t^k) \stackrel{\text{def}}{=} (x_{\min} + j\Delta x, k\Delta t). \tag{7}$$

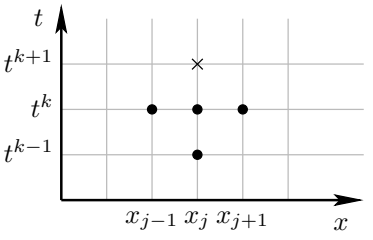


Fig. 1. Three-point scheme: u_j^{k+1} ($\times$) depends on u_{j-1}^k, u_j^k, u_{j+1}^k and u_j^{k-1} ($\bullet$)

Let v_h be a discrete function over $[0..j_{\max}] \times [0..k_{\max}]$. For all k in $[0..k_{\max}]$, we write $v_h^k = (v_j^k)_{0 \le j \le j_{\max}}$, then $v_h = ((v_h^k)_{0 \le k \le k_{\max}})$. A function v defined over Ω is approximated at the points of the grid by the discrete function v_h defined on $[0..j_{\max}] \times [0..k_{\max}]$ by $v_j^k \stackrel{\text{def}}{=} v(\mathbf{x}_j^k)$, except for u where we use the notation $\bar{u}_j^k \stackrel{\text{def}}{=} u(\mathbf{x}_j^k)$ to prevent notation clashes.

Let u_{0h} and u_{1h} be two discrete functions over $[0..j_{\max}]$; let s_h be a discrete function over $[0..j_{\max}] \times [0..k_{\max}]$. Then, the discrete function u_h over $[0..j_{\max}] \times [0..k_{\max}]$ is said to be the solution of the three-point[2] finite difference scheme, as illustrated in Figure 1, when the following set of equations holds:

$$\forall k \in [2..k_{\max}],\ \forall j \in [0..j_{\max}],$$

$$(L_h(c)\, u_h)_j^k \stackrel{\text{def}}{=} \frac{u_j^k - 2u_j^{k-1} + u_j^{k-2}}{\Delta t^2} + (A_h(c)\, u_h^{k-1})_j = s_j^{k-1}, \quad (8)$$

$$\forall j \in [0..j_{\max}],\ \ (L_{1h}(c)\, u_h)_j \stackrel{\text{def}}{=} \frac{u_j^1 - u_j^0}{\Delta t} + \frac{\Delta t}{2}(A_h(c)\, u_h^0)_j = u_{1,j}, \quad (9)$$

$$\forall j \in [0..j_{\max}],\ \ (L_{0h}\, u_h)_j \stackrel{\text{def}}{=} u_j^0 = u_{0,j}, \quad (10)$$

$$\forall k \in [0..k_{\max}[,\ \ u_{-1}^k = u_{j_{\max}+1}^k = 0 \quad (11)$$

where the matrix $A_h(c)$, discrete analog of $A(c)$, is defined, for any vector $v_h = ((v_j)_{0 \le j \le k_{\max}})$, by

$$\forall j \in [0..j_{\max}], \quad (A_h(c)\, v_h)_j \stackrel{\text{def}}{=} -c^2 \frac{v_{j+1} - 2v_j + v_{j-1}}{\Delta x^2}. \quad (12)$$

Note that defining u_h for artificial indexes $j = -1$ and $j = j_{\max} + 1$ is a trick to make the three-point spatial scheme valid for $j = 0$ and $j = j_{\max}$.

A discrete analog of the energy is also defined by[3]

$$E_h(c)(u_h)^{k+\frac{1}{2}} \stackrel{\text{def}}{=} \frac{1}{2} \left\| \frac{u_h^{k+1} - u_h^k}{\Delta t} \right\|_{\Delta x}^2 + \frac{1}{2} \left\langle u_h^k, u_h^{k+1} \right\rangle_{A_h(c)} \quad (13)$$

[2] In the sense "three spatial points", for the definition of matrix $A_h(c)$.

[3] By convention, the energy is defined between steps k and $k+1$, thus the notation $k + \frac{1}{2}$.

where $\langle v_h, w_h \rangle_{\Delta x} \stackrel{\text{def}}{=} \sum_{j=0}^{j_{\max}} v_j w_j \Delta x$, $\|v_h\|^2_{\Delta x} \stackrel{\text{def}}{=} \langle v_h, v_h \rangle_{\Delta x}$, and $\langle v_h, w_h \rangle_{A_h(c)} \stackrel{\text{def}}{=} \langle A_h(c)\, v_h, w_h \rangle_{\Delta x}$.

Note that the three-point scheme is parametrized by the discrete Cauchy data u_{0h} and u_{1h}, and by the discrete source term s_h. Of course, when u_{0h}, u_{1h}, and s_h are respectively approximations of u_0, u_1, f, then the discrete solution u_h is an approximation of the continuous solution u.

2.3 Convergence

Let ζ and ξ be in $]0,1[$ with $\zeta \leq 1 - \xi$. The CFL(ζ, ξ) condition (for Courant-Friedrichs-Lewy, see [19]) states that the discretization steps satisfy the relation

$$\zeta \leq \frac{c\Delta t}{\Delta x} \leq 1 - \xi. \tag{14}$$

Note that the lower bound ζ may seem surprising from a numerical analysis point of view; the formalization has however shown that it was mandatory (see Section 4.3).

The convergence error e_h measures the distance between the continuous and discrete solutions. It is defined by

$$\forall k \in [0..k_{\max}],\ \forall j \in [0..j_{\max}], \quad e_j^k \stackrel{\text{def}}{=} \bar{u}_j^k - u_j^k. \tag{15}$$

The truncation error ε_h measures at which precision the continuous solution satisfies the numerical scheme. It is defined by

$$\forall k \in [2..k_{\max}],\ \forall j \in [0..j_{\max}],\ \varepsilon_j^{k-1} \stackrel{\text{def}}{=} (L_h(c)\, \bar{u}_h)_j^k - s_j^{k-1}, \tag{16}$$

$$\forall j \in [0..j_{\max}],\ \varepsilon_j^0 \stackrel{\text{def}}{=} (L_{1h}(c)\, \bar{u}_h)_j - u_{1,j}, \tag{17}$$

$$\forall j \in [0..j_{\max}],\ \varepsilon_j^{-1} \stackrel{\text{def}}{=} (L_{0h} \bar{u}_h)_j - u_{0,j}. \tag{18}$$

The numerical scheme is said to be convergent of order 2 if the convergence error tends toward zero at least as fast as $\Delta x^2 + \Delta t^2$ when both discretization steps tend toward 0. More precisely, the numerical scheme is said to be convergent of order (p,q) uniformly on the interval $[0, t_{\max}]$ if the convergence error satisfies (see Section 3.2 for the definition of the big O notation that will be uniform with respect to space and time)

$$\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^p + \Delta t^q). \tag{19}$$

The numerical scheme is said to be consistent with the continuous problem at order 2 if the truncation error tends toward zero at least as fast as $\Delta x^2 + \Delta t^2$ when the discretization steps tend toward 0. More precisely, the numerical scheme is said to be consistent with the continuous problem at order $(p,\ q)$ uniformly on interval $[0, t_{\max}]$ if the truncation error satisfies

$$\left\| \varepsilon_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{[0,t_{\max}]}(\Delta x^p + \Delta t^q). \tag{20}$$

The numerical scheme is said to be stable if the discrete solution of the associated homogeneous problem (*i.e.* without any source term, $s(x,t)=0$) is bounded from above independently of the discretization steps. More precisely, the numerical scheme is said to be stable uniformly on interval $[0,t_{\max}]$ if the discrete solution of the problem without any source term satisfies

$$\exists \alpha, C_1, C_2 > 0,\ \forall t \in [0,t_{\max}],\ \forall \Delta x, \Delta t > 0, \quad \sqrt{\Delta x^2 + \Delta t^2} < \alpha \Rightarrow \left\| u_h^{k_{\Delta t}(t)} \right\|_{\Delta x} \leq (C_1 + C_2 t)(\|u_{0h}\|_{\Delta x} + \|u_{0h}\|_{A_h(c)} + \|u_{1h}\|_{\Delta x}). \quad (21)$$

The result to be formally proved at section 4 states that if the continuous solution u is regular enough on Ω and if the discretization steps satisfy the CFL $(\zeta,\ \xi)$ condition, then the three-point scheme is convergent of order (2, 2) uniformly on interval $[0,t_{\max}]$.

We do not admit (nor prove) the Lax equivalence theorem which stipulates that for a wide variety of problems and numerical schemes, consistency implies the equivalence between stability and convergence. Instead, we establish that consistency and stability implies convergence in the particular case of the one-dimensional acoustic wave equation.

3 The Coq Formalization: Basic Blocks

We decided to use the Coq proof assistant [20], as Coq was already used to prove the floating-point error [21] of this case study. All our developments use the Coq real standard (classical) library. Numerical equations, numerical schemes, numerical approximations deal with classical statements, and are not in the scope of intuitionist theory.

3.1 Dot Product

The function space $\mathbb{Z} \to \mathbb{R}$ can be equipped with pointwise addition and multiplication by a scalar. The result is a vector space. In the following, we are only interested in functions with finite support, that is the subset

$$F \stackrel{\text{def}}{=} \{f : \mathbb{Z} \to \mathbb{R} \mid \exists a, b \in \mathbb{Z}, \forall i \in \mathbb{Z}, f(i) \neq 0 \Rightarrow a \leq i \leq b\},$$

which is also a vector space. Then it is possible to define a dot product on F, noted $\langle .,. \rangle$, as follows:

$$\langle f, g \rangle \stackrel{\text{def}}{=} \sum_{i \in \mathbb{Z}} f(i)g(i) \quad (22)$$

and the corresponding norm $\|f\| \stackrel{\text{def}}{=} \sqrt{\langle f, f \rangle}$. The corresponding Coq formalization is not immediate, though. One could characterize F with a dependent type, but that would make operation $\langle .,. \rangle$ difficult to use (each time it is applied, proofs of finite support properties have to be passed as well). Instead, we define

$\langle ., . \rangle$ on the full function space $\mathbb{Z} \to \mathbb{R}$ using Hilbert's ε-operator (provided in Coq standard library in module `Epsilon`), as follows:

$$\langle f, g \rangle \stackrel{\text{def}}{=} \varepsilon \left(\begin{array}{c} \lambda x. \exists a\, b,\, (\forall i, (f(i) \neq 0 \vee g(i) \neq 0) \Rightarrow a \leq i \leq b) \\ \wedge\ x = \sum_{i=a}^{b} f(i) g(i) \end{array} \right) \tag{23}$$

Said otherwise, we give $\langle f, g \rangle$ a definition as a finite sum whenever f and g both have finite support and we let $\langle f, g \rangle$ undefined otherwise.

To ease the manipulation of functions with finite support, we introduce the following predicate characterizing such functions

$$FS(f) \stackrel{\text{def}}{=} \exists a\, b, \forall i, f(i) \neq 0 \Rightarrow a \leq i \leq b$$

and we prove several lemmas about it, such as

$$\begin{aligned} &\forall f g, FS(f) \Rightarrow FS(g) \Rightarrow FS(f + g) \\ &\forall f c, FS(f) \Rightarrow FS(c \cdot f) \\ &\forall f k, FS(f) \Rightarrow FS(i \mapsto f(i + k)) \end{aligned}$$

We also provide a Coq tactic to automatically discharge most goals about $FS(.)$. Finally, we can establish lemmas about the dot product, provided functions have finite support. Here are some of these lemmas:

$$\begin{aligned} &\forall f\, g\, c, FS(f) \Rightarrow FS(g) \Rightarrow \langle c \cdot f, g \rangle = c \cdot \langle f, g \rangle \\ &\forall f_1\, f_2\, g, FS(f_1) \Rightarrow FS(f_2) \Rightarrow FS(g) \Rightarrow \langle f_1 + f_2, g \rangle = \langle f_1, g \rangle + \langle f_2, g \rangle \\ &\forall f\, g, FS(f) \Rightarrow FS(g) \Rightarrow |\langle f, g \rangle| \leq \|f\| \cdot \|g\| \\ &\forall f\, g, FS(f) \Rightarrow FS(g) \Rightarrow \|f + g\| \leq \|f\| + \|g\| \end{aligned}$$

These lemmas are proved by reduction to finite sums, thanks to Formula (23). Note that the value of $\langle f, g \rangle_{\Delta x}$ defined in Section 2.2 is equal to $\Delta x \cdot \langle f, g \rangle$.

3.2 Big O Notation

For two functions f and g over $\mathbb{R}^n$, one usually writes $f(\boldsymbol{x}) = O_{\|\boldsymbol{x}\| \to 0}(g(\boldsymbol{x}))$ for

$$\exists \alpha, C > 0, \quad \forall \boldsymbol{x} \in \mathbb{R}^n, \quad \|\boldsymbol{x}\| \leq \alpha \Rightarrow |f(\boldsymbol{x})| \leq C \cdot |g(\boldsymbol{x})|.$$

Unfortunately, this definition is not sufficient for our formalism. Indeed, while $f(\mathbf{x}, \mathbf{\Delta x})$ will be defined over $\mathbb{R}^2 \times \mathbb{R}^2$, $g(\mathbf{\Delta x})$ will be defined over $\mathbb{R}^2$ only. So it begs the question: what to do about $\mathbf{x}$?

Our first approach was to use

$$\forall \mathbf{x}, \quad f(\mathbf{x}, \mathbf{\Delta x}) = O_{\|\mathbf{\Delta x}\| \to 0}(g(\mathbf{\Delta x}))$$

that is to say

$$\forall \mathbf{x}, \exists \alpha, C > 0, \quad \forall \mathbf{\Delta x} \in \mathbb{R}^2, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|$$

which means that α and C are functions of $\mathbf{x}$. So we would need to take the minimum of all the possible values of α, and the maximum for C. Potentially, they may be 0 and $+\infty$ respectively, making them useless.

In order to solve this issue, we had to define a notion of big O uniform with respect to the additional variable $\mathbf{x}$:

$$\exists \alpha, C > 0, \quad \forall \mathbf{x}, \mathbf{\Delta x}, \quad \|\mathbf{\Delta x}\| \leq \alpha \Rightarrow |f(\mathbf{x}, \mathbf{\Delta x})| \leq C \cdot |g(\mathbf{\Delta x})|.$$

Variables $\mathbf{x}$ and $\mathbf{\Delta x}$ are restricted to subsets S and P of $\mathbb{R}^2$. For instance, the big O that appears in Equation (19) uses

$$S = \mathbb{R} \times [0, t_{\max}],$$
$$P = \left\{ \mathbf{\Delta x} = (\Delta x, \Delta t) \mid 0 < \Delta x \ \wedge \ 0 < \Delta t \ \wedge \ \zeta \leq \frac{c \cdot \Delta t}{\Delta x} \leq 1 - \xi \right\}.$$

As often, the formal specification has allowed us to detect some flaws in usual mathematical pen-and-paper proofs, such as an erroneous switching of the universal and existential quantifiers hidden in the big O definition.

3.3 Taylor Expansion

The formalization assumes that "sufficiently regular" functions can be uniformly approximated by multivariate Taylor series. More precisely, the development starts by assuming that there exists two operators `partial_derive_firstvar` and `_secondvar`. Given a real-valued function f defined on the 2D plane and a point of it, they respectively return the functions $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial t}$ for this point, if they exist.

Again, these operators are similar to the use of Hilbert's ε operator. For documentation purpose, one could add two axioms stating that the returned function computes the derivatives for derivable functions; they are not needed for the later development though. Indeed, none of our proofs depend on the actual properties of derivatives; they only care about the fact that differential operators appear in both the regularity definition below and the wave equation.

The two primitive operators $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial t}$ are encompassed in a generalized differential operator $\frac{\partial^{m+n}}{\partial x^m \partial t^n}$. This allows us to define the 2D Taylor expansion of order n of a function f:

$$\mathrm{DL}_n(f, \mathbf{x}) \stackrel{\text{def}}{=} (\Delta x, \Delta t) \mapsto \sum_{p=0}^{n} \frac{1}{p!} \left(\sum_{m=0}^{p} \binom{p}{m} \cdot \frac{\partial^p f}{\partial x^m \partial t^{p-m}}(\mathbf{x}) \cdot \Delta x^m \cdot \Delta t^{p-m} \right).$$

A function f is then said to be sufficiently regular of order n if

$$\forall m \leq n, \quad \mathrm{DL}_{m-1}(f, \mathbf{x})(\mathbf{\Delta x}) - f(\mathbf{x} + \mathbf{\Delta x}) = O\left(\|\mathbf{\Delta x}\|^m\right). \tag{24}$$

4 The Coq Formalization: Convergence

4.1 Wave Equation

As explained in Section 2, a solution of the wave equation with given u_0, u_1 and s verifies Equations (1)–(3). Its discrete approximation verifies Equations (8)–(10). Both are directly translated in Coq using the definitions of Section 3. Concerning the discretization, we choose that the space index is in $\mathbb{Z}$ (to be coherent with the dot product definition of Section 3.1) while the time index is in $\mathbb{N}$.

Our goal is to prove the uniform convergence of the scheme with order (2,2) on the interval $[0, t_{\max}]$:

$$\left\| e_h^{k_{\Delta t}(t)} \right\|_{\Delta x} = O_{\substack{t \in [0,\, t_{\max}] \\ (\Delta x, \Delta t) \to 0 \\ 0 < \Delta x \,\wedge\, 0 < \Delta t \,\wedge \\ \zeta \le c\frac{\Delta t}{\Delta x} \le 1 - \xi}} (\Delta x^2 + \Delta t^2).$$

4.2 Finite Support

The proofs concerning the convergence of the scheme rely on the dot product. As explained in Section 3.1, the dot product requires the functions to have a finite support in order to apply any lemma. We therefore proved the finiteness of the support of many functions. We assume that the inputs u_0, u_1, and s of the wave equation have a finite support. More precisely, we assume that there exists χ_1 and χ_2 such that $u_0(x) = u_1(x) = 0$ for all x out of $[\chi_1, \chi_2]$ and $s(x,t) = 0$ for all x out of $[\chi_1 - c \cdot t, \chi_2 + c \cdot t]$ where c is the velocity of propagation of waves in Equation (1).

Figure 2 describes the nullity, that is to say the finite support, of the various functions. We needed to prove the finiteness of their support:

- u_0 and u_1 by hypothesis and therefore $u_{0,j}$ and $u_{1,j}$.
- s (for any value t) by hypothesis and therefore s_j^k is zero outside of a cone of slope c^{-1}.

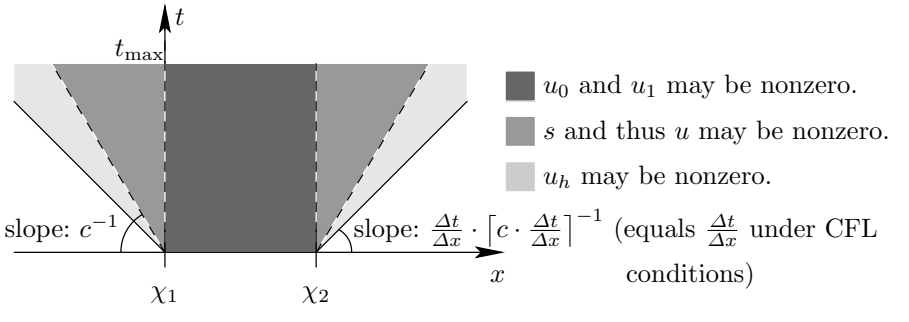


Fig. 2. Finite supports. The support of the Cauchy data u_0 and u_1 is included in the support of the continuous source term s, and of the continuous solution u. Which is in turn also included in the support of the discrete solution u_h, provided that the CFL condition holds. For a finite $t_{\max}$, all these supports are finite.

- the scheme itself has a finite support: due to the definition of u_j^k and the nullity of $u_{0,j}$ and $u_{1,j}$ and s_j^k, we can prove that u_j^k is zero outside of a cone of slope $\frac{\Delta t}{\Delta x} \cdot \left\lceil c \cdot \frac{\Delta t}{\Delta x} \right\rceil^{-1}$. Under CFL$(\zeta, \xi)$ conditions, this slope will be $\frac{\Delta t}{\Delta x}$.
- the truncation and convergence errors also have finite support with the previous slope.

We need here an axiom about the nullity of the continuous solution. We assume that the continuous solution $u(x,t)$ is zero for x out of $[\chi_1 - c \cdot t, \chi_2 + c \cdot t]$ (same as s). This is mathematically correct, since it derives from d'Alembert's formula (6). But its proof is out of the scope of the current formalization and we therefore preferred to simply add the nullity axiom.

4.3 Consistency

We first prove that the truncation error is of order $\Delta x^2 + \Delta t^2$. The idea is to show that, for $\mathbf{\Delta x}$ small enough, the values of the scheme L_h are near the corresponding values of L. This is done using the properties of Taylor expansions. This involves long and complex expressions but the proof is straightforward.

We first prove that the truncation error in one point (j,k) is a $O(\Delta x^2 + \Delta t^2)$. This is proved for $k = 0$ and $k = 1$ by taking advantage of the initializations and Taylor expansions. For bigger k, the truncation error reduces to the sum of two Taylor expansions of degree 3 in time (this means $m = 4$ in Formula (24)) and two Taylor expansions of degree 3 in space that partially cancel (divided by something proportional to $\|\mathbf{\Delta x}\|^2$). Here, we take advantage of the generality of big O as we consider the sum of a Taylor expansion on Δx and of a Taylor expansion on $-\Delta x$. If we had required $0 < \Delta x$ (as a space grid step), we could not have done this proof.

The most interesting part is to go from pointwise consistency to uniform consistency. We want to prove that the norm of the truncation error (in the sense of the infinite dot product $\langle \cdot, \cdot \rangle_{\Delta x}$) is also $O(\Delta x^2 + \Delta t^2)$. We therefore need to bound the number of nonzero values of the truncation error. As explained in Section 4.2, the truncation error values at time $k \cdot \Delta t$ may be nonzero between ${\chi_1}'_k = \left\lfloor \frac{\chi_1}{\Delta x} \right\rfloor - \left\lceil c \cdot \frac{\Delta t}{\Delta x} \right\rceil k$ and ${\chi_2}'_k = \left\lceil \frac{\chi_2}{\Delta x} \right\rceil + \left\lceil c \cdot \frac{\Delta t}{\Delta x} \right\rceil k$. This gives a number of terms N roughly bounded by (all details are handled in the formal proof):

$$
\begin{aligned}
N &\leq \frac{{\chi_2}'_k - {\chi_1}'_k}{\Delta x} \leq \frac{\chi_2 - \chi_1}{\Delta x^2} + 2 \cdot k_{\max} \cdot \frac{\left\lceil c \cdot \frac{\Delta t}{\Delta x} \right\rceil}{\Delta x} \\
&\leq \frac{\chi_2 - \chi_1}{\Delta x^2} + 2 \cdot \frac{t_{\max}}{\Delta t} \cdot \frac{c \cdot \frac{\Delta t}{\Delta x} + 1}{\Delta x}
\end{aligned}
$$

As the norm is a Δx-norm, this reduces to bounding with a constant value the value $N \cdot \Delta x^2$ which is smaller than $\chi_2 - \chi_1 + 2 \cdot t_{\max} \cdot c + 2 \cdot t_{\max} \cdot \frac{\Delta x}{\Delta t}$. To bound this with a constant value, we require $c\frac{\Delta t}{\Delta x}$ to have a constant lower bound ζ (it already had an upper bound $1 - \xi$). Then $N \cdot \Delta x^2 \leq \chi_2 - \chi_1 + 2 \cdot t_{\max} \cdot c + 2 \cdot c \cdot t_{\max} \cdot \frac{1}{\zeta}$ which is constant.

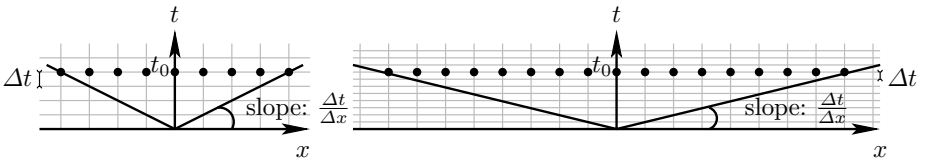


Fig. 3. For a given time t_0, the number of nonzero values increases when the slope $\frac{\Delta t}{\Delta x}$ goes to zero. From left to right, Δt is divided by 2 whereas Δx remains the same. We can see that the number of nonzero terms is almost doubled (from 9 to 17).

Mathematically, this requirement comes as a surprise. The following scenario explains it. If $c\frac{\Delta t}{\Delta x}$ goes to zero, then Δt goes to zero much faster than Δx. It corresponds to Figure 3. The number of nonzero terms (for u_h and thus for the truncation error) goes to infinity as $\frac{\Delta t}{\Delta x}$ goes to zero.

4.4 Stability

To prove stability, we use the discrete energy defined in Equation (13). From the properties of the scheme, we calculate the evolution of the energy. At each step, it increases by a known value. In particular, if s is zero, the discrete energy (as the continuous energy) is constant:

$$\forall k > 0, \quad E_h(c)(u_h)^{k+\frac{1}{2}} - E_h(c)(u_h)^{k-\frac{1}{2}} = \frac{1}{2}\left\langle u_h^{k+1} - u_h^{k-1}, s_h^k \right\rangle_{\Delta x}.$$

From this, we give an underestimation of the energy:

$$\forall k, \quad \frac{1}{2}\left(1 - \left(c\frac{\Delta t}{\Delta x}\right)^2\right)\left\|\frac{u_h^{k+1} - u_h^k}{\Delta t}\right\|_{\Delta x} \leq E_h(c)(u_h)^{k+\frac{1}{2}}.$$

Therefore we have the nonnegativity of the energy under CFL(ζ, ξ) conditions. For convergence, the key result is the overestimation of the energy:

$$\sqrt{E_h(c)(u_h)^{k+\frac{1}{2}}} \leq \sqrt{E_h(c)(u_h)^{\frac{1}{2}}} + \frac{\sqrt{2}}{2\sqrt{2\xi - \xi^2}} \cdot \Delta t \cdot \sum_{j=1}^{k} \|i \mapsto s_h(i,j)\|_{\Delta x}$$

for all time t, with $k = \left\lfloor \frac{t}{\Delta t} \right\rfloor - 1$.

This completes the stability proof. In the inequality above, the energy is bounded for u_h, but the bound is actually valid for all the solutions of the discrete scheme, for any initial conditions and source term.

Note that the formal proof of stability closely follows the mathematical pen-and-paper proof and no additional hypotheses were found to be necessary.

4.5 Convergence

We prove that the convergence error is the solution of a scheme and therefore the results of Section 4.4 apply to it. More precisely, for all $\mathbf{\Delta x}$, the convergence error is solution of a discrete scheme with inputs

$$u_{0,j} = 0, \qquad u_{1,j} = \frac{e_j^1}{\Delta t}, \qquad \text{and} \qquad s_j^k = \varepsilon_j^{k+1},$$

where the errors refer to the errors of the initial scheme of the wave equation with grid steps $\mathbf{\Delta x}$. (Actual Coq notations depend on many more variables.)

We have proved many lemmas about the initializations of our scheme and of the convergence error. The idea is to prove that the initializations of the scheme are precise enough to guarantee that the initial convergence errors (at step 0 and 1) are accurate enough.

We also bounded the energy of the convergence error. Using results of Section 4.4, the proof reduces to bounding the sum of the source terms, here the truncation errors. Using results of Section 4.3, we prove this sum to be $O(\Delta x^2 + \Delta t^2)$. A few more steps conclude the proof.

Once more, the formal proof follows the pen-and-paper proof and progresses smoothly under the required hypothesis, including all the conditions on $\frac{\Delta t}{\Delta x}$ of Equation (14).

5 Conclusion and Perspectives

One of the goals of this work is to favor the use of formal methods in numerical analysis. It may seem to be just wishful thinking, but it is actually seen as needed by some applied mathematicians. An early case led to the certification of the O∂yssée tool [10]. This tool performs automatic differentiation, which is one of the basic blocks for *gradient*-based algorithms. Our work tackles the converse problem: instead of considering derivation-based algorithms, we have formalized and proved part of the mathematical background behind integration-based algorithms.

This work shows there may be a synergy between applied mathematicians and logicians. Both domains are required here: applied mathematics for an initial proof that could be enriched upon request and formal methods for machine-checking it. This may be the reason why such proofs are scarce as this kind of collaboration is uncommon.

Proof assistants seem to mainly deal with algebra, but we have demonstrated that formalizing numerical analysis is possible too. We can confirm that pen-and-paper proofs are sometimes sketchy: they may be fuzzy about the needed hypotheses, especially when switching quantifiers. We have also learned that filling the gaps may cause us to go back to the drawing board and to change the basic blocks of our formalization to make them more generic (a big O that needs to be uniform and also generic with respect to a property P).

The formal bound on the error method, while of mathematical interest, is not sufficient to guarantee the correction of numerical applications implementing the three-point scheme. Indeed, such applications usually perform approximated computations, *e.g.*, floating-point computations, for efficiency and simplicity reasons. As a consequence, the proof of the method error has to be combined with a proof on the rounding error, in order to get a full-fledged correction proof. Fortunately, the proof on the rounding error has already been achieved [21]. We

are therefore close to having a formal proof of both the numerical scheme and its floating-point implementation.

An advantage of Coq with respect to most other proof assistants is the ability to *extract* programs from proofs [22]. For this work, it does not make much sense to extract the algorithm from the proofs: not only is the algorithm already well-known, but its floating-point implementation was also certified [21]. So, an extraction of the algorithm would not bring much. However, extraction gives access to the constant C hidden behind the big O notation. Indeed, the proof of the floating-point algorithm relies on the discrete solution being good enough, so that the computed result does not diverge. Precisely, the convergence error has to be smaller than 1, and an extracted computation would be able to ensure this property. Furthermore, having access to this constant can be useful to the applied mathematicians for the a posteriori estimations needed for adaptive mesh refinements. Extraction also gives access to the α constant. That way, we could check that the constant $\mathbf{\Delta x}$ chosen in the C program described in [21] verifies this requirement. Note that performing an extraction requires to modify the definition of the big O so that it lives in `Set` instead of `Prop`. But this formalization change happens to be straightforward and Coq then succeeds in extracting mathematical formulas for constants α and C. Only basic operators (*e.g.* $+$, $\sqrt{\cdot}$, min) and constants (*e.g.* $t_{\max}$, ξ, χ_1, Taylor constants) appear in them, so they should be usable in practice.

The formal development is about 4500-line long. Its dependency graph is detailed in Figure 4. About half of the development is a reusable library described in Section 3 and the other half is the proof of convergence of the numerical scheme described in Section 4. This may seem a long proof for a single scheme for a single PDE. To put it into perspective, usual pen-and-paper proofs are 10-page long and an in-depth proof can be 60-page long. (We wrote one to ensure that we were not getting sidetracked.) So, at least from a length point of view, the formal proof is comparable to a detailed pen-and-paper proof.

In the end, the whole development contains only two axioms: the ε operator for the infinite dot product (see Section 3.1) and the finite support of the continuous solution of the wave equation (see Section 4.2). So, except for this last axiom which is related to the chosen PDE, the full numerical analysis proof of convergence is machine-checked and all required hypotheses are made clear. There is no loss of confidence due to this axiom, since the kind of proof and the results it is based upon are completely different from the ones presented here. Indeed, this axiom is about continuous solutions and hence much less error-prone.

For this exploratory work, we only considered the simple three-point scheme for the one-dimensional wave equation. Further works involve generalizing our approach to other schemes and other PDEs. We are confident that it would scale to higher-dimension and higher-order equations solved by discrete numerical schemes. However, the proofs of Section 4 are entangled with particulars of the presented problem, and would therefore have to be redone for other problems. So a more fruitful approach would be to prove once and for all the Lax equivalence theorem that states that consistency implies the equivalence between

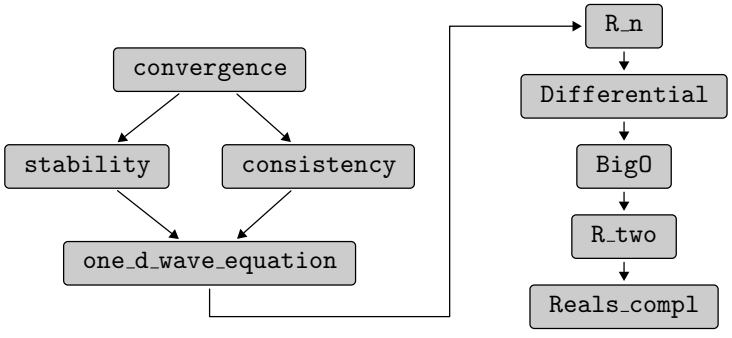


Fig. 4. Dependency graph of the Coq development. On the left are the files from the convergence proof. The other files correspond to the reusable library.

convergence and stability. This would considerably reduce the amount of work needed for tackling other schemes and equations.

This work also showed us that summations and finite support functions play a much more important role in the development than we first expected. We are therefore considering moving to the SSReflect interface and libraries for Coq [23], so as to simplify the manipulations of these objects in our forthcoming works.

References

1. Thomas, J.W.: Numerical Partial Differential Equations: Finite Difference Methods. Texts in Applied Mathematics, vol. 22. Springer, Heidelberg (1995)
2. Zwillinger, D.: Handbook of Differential Equations. Academic Press, London (1998)
3. Dutertre, B.: Elements of Mathematical Analysis in PVS. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 141–156. Springer, Heidelberg (1996)
4. Harrison, J.: Theorem Proving with the Real Numbers. Springer, Heidelberg (1998)
5. Fleuriot, J.D.: 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)
6. Mayero, M.: Formalisation et automatisation de preuves en analyses réelle et numérique. PhD thesis, Université Paris VI (2001)
7. Gamboa, R., Kaufmann, M.: Nonstandard Analysis in ACL2. Journal of Automated Reasoning 27(4), 323–351 (2001)
8. Geuvers, H., Niqui, M.: Constructive Reals in Coq: Axioms and Categoricity. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 79–95. Springer, Heidelberg (2002)
9. Cruz-Filipe, L.: A Constructive Formalization of the Fundamental Theorem of Calculus. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 108–126. Springer, Heidelberg (2003)
10. Mayero, M.: Using Theorem Proving for Numerical Analysis (Correctness Proof of an Automatic Differentiation Algorithm). In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 246–262. Springer, Heidelberg (2002)

11. Harrison, J.: A HOL Theory of Euclidean Space. In: Hurd, J., Melham, T.F. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
12. Avigad, J., Donnelly, K.: A Decision Procedure for Linear "Big O" Equations. J. Autom. Reason. 38(4), 353–373 (2007)
13. Bécache, E.: Étude de schémas numériques pour la résolution de l'équation des ondes. Master Modélisation et simulation, Cours ENSTA (2009), http://www-rocq.inria.fr/~becache/COURS-ONDES/Poly-num-0209.pdf
14. Achenbach, J.D.: Wave Propagation in Elastic Solids. North Holland, Amsterdam (1973)
15. Brekhovskikh, L.M., Goncharov, V.: Mechanics of Continua and Wave Dynamics. Springer, Heidelberg (1994)
16. Newton, I.: Axiomata, sive Leges Motus. In: Philosophiae Naturalis Principia Mathematica, London, vol. 1 (1687)
17. le Rond D'Alembert, J.: Recherches sur la courbe que forme une corde tendue mise en vibrations. In: Histoire de l'Académie Royale des Sciences et Belles Lettres (Année 1747), vol. 3, pp. 214–249. Haude et Spener, Berlin (1749)
18. John, F.: Partial Differential Equations. Springer, Heidelberg (1986)
19. Courant, R., Friedrichs, K., Lewy, H.: On the partial difference equations of mathematical physics. IBM Journal of Research and Development 11(2), 215–234 (1967)
20. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
21. Boldo, S.: Floats & Ropes: a case study for formal numerical program verification. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 91–102. Springer, Heidelberg (2009)
22. Letouzey, P.: A New Extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
23. Bertot, Y., Gonthier, G., Ould Biha, S., 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)

An Efficient Coq Tactic for Deciding Kleene Algebras*

Thomas Braibant and Damien Pous

LIG, UMR 5217, CNRS – INRIA

Abstract. We present a reflexive tactic for deciding the equational theory of Kleene algebras in the Coq proof assistant. This tactic relies on a careful implementation of efficient finite automata algorithms, so that it solves casual equations almost instantaneously. The corresponding decision procedure was proved correct and complete; correctness is established w.r.t. any model (including binary relations), by formalising Kozen's initiality theorem.

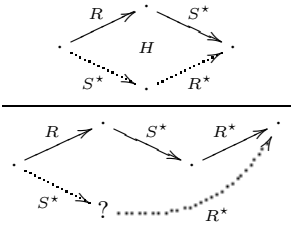
Motivations

Proof *assistants* like Coq or Isabelle/HOL make it possible to leave technical or administrative details to the computer, by defining high-level tactics. For example, one can define tactics to solve decidable problems automatically (e.g., `omega` for Presburger arithmetic and `ring` for ring equalities). Here we present a tactic for solving equations and inequations in Kleene algebras. This tactic belongs to a larger project whose aim is to provide tools for working with binary relations in Coq. Indeed, Kleene algebras correspond to a non-trivial decidable fragment of binary relations. In the long term, we plan to use these tools to formalise results in rewriting theory, process algebras, and concurrency theory results. Binary relations play a central role in the corresponding semantics.

A starting point for this work is the following remark: proofs about abstract rewriting (e.g., Newman's Lemma, equivalence between weak confluence and the Church-Rosser property, termination theorems based on commutation properties) are best presented using informal "diagram chasing arguments". This is illustrated by Fig. 1, where the same state of a typical proof is represented three times. Informal diagrams are drawn on the left. The goal listed in the middle corresponds to a naive formalisation where the points related by relations are mentioned explicitly. This is not satisfactory: a lot of variables have to be introduced, the goal is displayed in a rather verbose way, the user has to draw the intuitive diagrams on its own paper sheet. On the contrary, if we move to an algebraic setting (the right-hand side goal), where binary relations are seen as abstract objects that can be composed using various operators (e.g., union, intersection, relational composition, iteration), statements and Coq's output become rather compact, making the current goal easier to read and to reason about.

* A preliminary version of this work was presented at the 1st Coq Workshop, 2009.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 163–178, 2010.




```
R,S: relation P
H: ∀p,r,q, R p r → S* r q
    → ∃s, S* p s & R* s q
p,q,q',s: P
Hpq: R p q
Hqs: S* q s
Hsq': R* s q'
=========================
∃s: P, S* p s & R* s q'
```

```
R,S: X
H: R·S* ⊆ S*·R*
===============
R·S*·R* ⊆ S*·R*
```

Fig. 1. Diagrammatic, concrete, and abstract presentations of the same state in a proof

More importantly, moving to such an abstract setting allows us to implement several decision procedures, that could hardly be stated with the concrete presentation. For example, once we rewrite H in the right-hand side goal of Fig. 1, we obtain the inclusion $S^\star{\cdot}R^\star{\cdot}R^\star{\subseteq}S^\star{\cdot}R^\star$ which is a (straightforward) theorem of Kleene algebras: the tactic we describe in this paper proves it automatically.

Outline. We give some mathematical background and we sketch the overall structure of the tactic in Sect. 1. The underlying design choices are discussed in Sect. 2. We describe the algorithm and its correctness proof in Sect. 3. Section 4 is devoted to related works and directions for future work.

1 Deciding Equalities in Kleene Algebras

Theoretical background. A Kleene algebra [20] is a tuple $\langle X, \cdot, +, 1, 0, \star\rangle$, where $\langle X, \cdot, +, 1, 0\rangle$ is an idempotent non-commutative semiring, and $\star$ is a unary operation on X, satisfying the following axiom and inference rules (where $\leq$ is the preorder defined by $x \leq y \triangleq x + y = y$):

$$1 + a \cdot a^\star \leq a^\star \qquad \frac{a \cdot x \leq x}{a^\star \cdot x \leq x} \qquad \frac{x \cdot a \leq x}{x \cdot a^\star \leq x}$$

Terms of Kleene algebras are called *regular expressions*, irrespective of the considered model. Models of Kleene algebras include *regular languages*, where the star operation is language iteration; and *binary relations*, where the product $(\cdot)$ is relational composition, and star is reflexive and transitive closure. Thanks to finite automata theory [19,29], equality of regular languages is decidable:

> *"two regular languages are equal if and only if the corresponding minimal automata are isomorphic".*

However, the above theorem is not sufficient to decide equations in all Kleene algebras: it only applies to the regular languages model. We actually need a more recent theorem, by Kozen [20] (independently proved by Krob [24]):

> *"if two regular expressions α and β denote the same regular language, then $\alpha = \beta$ can be proved in any Kleene algebra".*

In other words, the algebra of regular languages is initial among Kleene algebras: we can use finite automata algorithms to solve equations in an arbitrary Kleene algebra $\mathcal{A}$. The main idea of Kozen's proof is to encode finite automata using matrices over $\mathcal{A}$, and to replay the algorithms at this algebraic level. Indeed, a finite automaton with transitions labelled by the elements of $\mathcal{A}$ can be represented with three matrices $\langle u, M, v\rangle \in \mathcal{M}_{1,n} \times \mathcal{M}_{n,n} \times \mathcal{M}_{n,1}$: n is the number of states of the automaton; u and v are 0-1 vectors respectively coding for the sets of initial and accepting states; and M is the transition matrix: $M_{i,j}$ labels transitions from state i to state j.

We remark that the product $u \cdot M \cdot v$ is a scalar (i.e., a regular expression), which can be thought of as the set of one-letter words accepted by the automaton. Therefore, to mimic the behaviour of a finite automaton, we just need to iterate over the matrix M. This is possible thanks to another theorem, which actually is the crux of the initiality theorem: *"square matrices over a Kleene algebra form a Kleene algebra"*. We hence have a star operation on matrices, and we can interpret an automaton algebraically, by considering the product $u \cdot M^\star \cdot v$.

Overview of our strategy. We define a *reflexive* tactic. This methodology is quite standard [2]: for example, this is how the `ring` tactic is implemented [14]. Concretely, this means that we implement the decision procedure as a Coq program, and that we prove its correctness and completeness within the proof assistant:

```
Definition decide_kleene: regex → regex → bool := ...
Theorem Kozen: ∀ a b, decide_kleene a b = true ↔ a == b.
```

The above statement corresponds to correctness and completeness with respect to the syntactic "free" Kleene algebra': `regex` is the inductive type for regular expressions over a given set of variables, and `==` is the inductive equality generated by the axioms of Kleene algebras and the rules of equational reasoning (see Fig. 4). Using reification mechanisms, this is sufficient for our needs: the result can be lifted to other models using simple tactics (see Sect. 2.3).

The equational theory of Kleene algebras is PSPACE-complete [26]; this means that the `decide_kleene` function must be written with care, using efficient out-of-the-shelf algorithms. Notably, the matricial representation of automata is not efficient, so that formalising Kozen's "mathematical" proof [20] in a naive way would be computationally impracticable. Instead, we need to chose appropriate data structures for automata and algorithms, and to rely on the matricial representation only in proofs, using the adequate translation functions.

2 Underlying Design Choices

Before giving more details about our implementation of the decision procedure (Sect. 3), we explain the main choices we made about the design of our library: definition of the algebraic hierarchy, representation of matrices and handling of heterogeneous structures, reification mechanism, and numbers representation.

2.1 Algebraic Hierarchy

The mathematical definition of a Kleene algebra is incremental: a Kleene algebra is a non-commutative semiring, which is itself composed of a monoid and a semilattice. Moreover, proofs naturally follow this hierarchy: when proving results about semirings, one usually relies on results about both monoids and semilattices. In order to structure our development in a similar way, we defined the algebraic hierarchy using Coq's recent *typeclasses* mechanism [30]: we defined several classes, corresponding to the different algebraic structures, so as to obtain the following "sub-class" relations:

```
SemiLattice <:
   Monoid   <: SemiRing <: KleeneAlgebra <: ...
```

Records vs. modules. Typeclasses are based on *records*; another possibility was to use *modules*. We tried the latter one; it was however quite difficult to organise modules, signatures, and functors so as to obtain the desired level of sharing between the various proofs. In particular, when we consider more complex algebraic structures, we can no longer work with syntactical sub-typing between structures (we only have functors from one structure to another) and we lose the ability to directly use theorems, definitions, and tactics from lower structures in higher structures. Except for some limitations due to the novelty of this feature, typeclasses happen to be much easier to use than modules for defining such a hierarchy. Sharing is obtained in a straightforward way, the code does not need to be written in a monolithic way (as opposed to using functors), and there are nice and simple solutions for overloading notations (e.g., we can use the same infix symbol for multiplication in a monoid, a semiring, or a matrix semiring).

Typeclasses vs. canonical structures. *Canonical structures* is another record-based inference mechanism (which we incidentally use to declare concrete structures). We tried to use canonical structures instead of typeclasses to define the algebraic hierarchy from the beginning, along the lines of [13,3,12]. The overall benefit was unclear. This is mainly because our hierarchy is not really deep, so that we can handle it with typeclasses without introducing a complex infrastructure (we reached the limit, however: some of our proofs are noticeably slow to compile, due to this simplistic approach). Another reason is that we lack a deep understanding of Coq internal unification algorithm, which currently seems to be required to work efficiently with canonical structures. Therefore, we might switch to canonical structures at some point, when this technology will be better understood and supported.

2.2 Matrices

A matrix can be seen as a partial map from pairs of integers to a given type X, so that a Coq type for matrices and a sum operation could be defined as follows:

```
Definition MX (n m: nat) := ∀ i j, i<n → j<m → X.
Definition plus n m (M N: MX n m) i j (Hi: i<n) (Hj: j<m) :=
    M i j Hi Hj + N i j Hi Hj.
```

This corresponds to the dependent types approach: a matrix is a map to X from two integers and two proofs that these integers are lower than the bounds of the matrix. Except for the concrete representation, this is the approach followed in [3,12,4]. With such a type, every access to a matrix element is made by exhibiting two proofs, to ensure that indices lie within the bounds. This is not problematic for simple operations like the above `plus` function; this however requires more boilerplate for other functions, like block decomposition operations.

We actually adopt another strategy: we move bounds checks to equality proofs, by working with the following definitions:

```
Definition MX n m := nat → nat → X.
Definition equal n m (M N: MX n m) := ∀ i j, i<n → j<m → M i j == N i j.
Fixpoint sum i k (f: nat → X) :=
    match k with 0 ⇒ 0 | S k ⇒ f i + sum (S i) k f end.
Definition dot n m p (M: MX n m) (N: MX m p) :=
    fun i j ⇒ sum 0 m (fun k ⇒ M i k · N k j).
```

Here, a matrix is an infinite function from pairs of integers to X, and equality is restricted to the domain of the matrix. With these definitions, we do not need to manipulate proofs when defining matrix operations (like the above `dot` function), so that subsequent definitions are easier to write. Bounds checks are required a posteriori only, when proving properties about these matrices operations, e.g., associativity of the product. This is generally straightforward: these proofs are done within the interactive proof mode, so that they can be solved with high level tactics like `omega`. (Note that this separation between proofs and programs could also be achieved syntactically—even with a dependently typed definition of matrices—by using Coq's `Program` feature.)

Although the correctness proof of our algorithm heavily relies on matricial reasoning, and in particular block matrix decompositions, we have not found major drawbacks to this approach yet. We actually believe that it would scale smoothly to even more intensive usages of matrices, e.g., linear algebra [12].

Phantom types. Unfortunately, these non-dependent definitions allow one to type the following code, where the three additional arguments of `dot` are implicit:

```
Definition ill_dot n p (M: MX n 16) (N: MX 64 p): MX n p := dot M N.
```

This definition is accepted because of the conversion rule: since the dependent type `MX n m` does not mention `n` nor `m` in its body, these type informations can be discarded by the type system using the conversion rule (`MX n 16 = MX n 64`). While such an ill-formed definition will be detected at proof-time; it is a bit sad not to benefit from the advantages of a strongly typed programming language here. We solved this problem at the cost of some syntactic sugar, by resorting to an inductive singleton definition, reifying bounds in *phantom types*:

```
Inductive MX (n m: nat) := box: (nat → nat → X) → MX n m.
Definition get (n m: nat) (M: MX n m) := match M with box f ⇒ f end.
Definition plus (n m: nat) (M N: MX n m) :=
    box n m (fun i j ⇒ get M i j + get N i j).
```

Coq no longer equates types `MX n 16` and `MX n 64` with this definition, so that the above `ill_dot` function is rejected, and we can trust inferred implicit arguments (e.g., the `m` argument of `dot`).

```
X: Type.

dot:  X → X → X.
one:  X.
plus: X → X → X.
zero: X.
star: X → X.

dot_neutral_left:
 ∀ x, dot one x == x.
...
```

```
T: Type.
X: T → T → Type.

dot:  ∀ n m p, X n m → X m p → X n p.
one:  ∀ n,     X n n.
plus: ∀ n m,   X n m → X n m → X n m.
zero: ∀ n m,   X n m.
star: ∀ n,     X n n → X n n.

dot_neutral_left:
 ∀ n m (x: X n m), dot one x == x.
...
```

Fig. 2. From Kleene algebras to typed Kleene algebras

Computation. Although we do not use matrices in computations in this work, we also advocate this lightweight representation from the efficiency point of view. First, using non-dependent types is more efficient: not a single boundary proof gets evaluated in matrix computations (e.g., matrix multiplications). Second, using functions to represent matrices is two-edged: on the one hand, if the matrix resulting of a computation is seldom used, then computing its elements by need is efficient; on the other hand, making numerous accesses to the same expensive computation may be a burden. To this end, we defined a *memoisation* operator that computes all elements of a given matrix, stores the results in a map, and returns the closure that looks up in the map rather than recomputing the result. This memoisation operator is proved to be an identity; it can be inserted in matrix computations in a transparent way, at judicious places.

```
Definition mx_force n m (M: MX n m): MX n m :=
  let l := mx_to_lists M in box n m (fun i j ⇒ nth i (nth j l)).
Lemma mx_force_id : ∀ n m (M : MX n m), mx_force M == M.
```

2.3 Typed Algebras, Typed Reification

Adding types. We need to work with *rectangular* matrices at several places: to prove the correctness of some steps of the algorithm (see Sect. 3.3), and to treat vectors as matrices so as to factorise proofs. However, while *square* matrices over a semiring form a semiring, this is not the case for *rectangular* matrices: the various operations are only partial, dimensions have to agree. Therefore, with naive definitions of the algebraic structures, we are unable to use theorems and tools developed for monoids, semi-lattices, and semirings to reason about rectangular matrices. To remedy this problem, we generalised algebraic structures from the beginning using *types*. An example is given in Fig. 2: a typical signature for semirings is presented on the left-hand side; we moved to the signature on the right-hand side, where a set `T` of indices (or types) is used to constrain the various operations. These abstract indices can be thought of as matrix dimensions; we actually moved to a categorical setting: `T` is a set of objects, `X n m` is the set of morphisms from `n` to `m`, `one` is the set of identities, and `dot` is composition.

As expected, with such definitions, one can form arbitrary matrices over a typed structure, and obtain another instance of this typed structure. In particular, the matrices over an arbitrary typed Kleene algebra form a typed Kleene

algebra. Then, thanks to typeclasses, we inherit all theorems, tactics, and notations we defined on generic structures, at the matricial level. Notably, when defining the star operation on matrices over a Kleene algebra, we can benefit from all tools for semirings, monoids, and semi-lattices. This is quite important since this construction is rather complicated.

Removing types. Typed structures not only make it easier to work with rectangular matrices, they also give rise to a wider range of models. In particular, we can consider heterogeneous binary relations rather than binary relations on a single fixed set. This leads to the following question: can the usual decision procedures be extended to this more general setting? Consider for example the equation $a \cdot (b \cdot a)^\star = (a \cdot b)^\star \cdot a$, which is a theorem of typed Kleene algebras as soon as a and b are respectively given types $n \to m$ and $m \to n$, for some n, m. How to ensure that the untyped automata algorithms respect types and actually give valid, well-typed, proofs?

For efficiency and practicability reasons, extending our decision procedure to work with typed elements is not an option. Instead, we proved the following theorem, which allows one to erase types, i.e., to transform a typed equality goal into an untyped one:

$$\frac{\vdash u = v \qquad \Gamma \vdash u \rhd \alpha : n \to m \qquad \Gamma \vdash v \rhd \beta : n \to m}{\vdash \alpha = \beta : n \to m} \qquad (*)$$

Here, $\Gamma \vdash u \rhd \alpha : n \to m$ reads "under the evaluation and typing context Γ, the untyped term u can be evaluated to α, of type $n \to m$"; this predicate can be defined inductively in a straightforward way, for various algebraic structures. The theorem can then be rephrased as follows: "if the untyped terms u and v are equal, then for all typed interpretations α and β of u and v, the typed equality $\alpha = \beta$ holds". See [28] for a theoretical study of these untyping theorems; also note that Kozen investigated a similar question [21] and came up with a slightly different solution: he solves the case of the Horn theory rather than equational theory, at the cost of working in a restrained form of Kleene algebras.

Typed reification. The above discussion about types raises another issue: reflexive tactics need to work with syntactical objects. For example, in order to construct an automaton, we need to proceed by structural induction on the given expression. This step is commonly achieved by moving to the free algebra of terms, and resorting to Coq's reification mechanism (`quote`). However, this mechanism does not handle typed structures, so that we needed to re-implement it. Since we do not have binders, we were able do this within `Ltac`: it suffices to `eapply` theorem $(*)$ to the current goal, so that we are left with three goals, with holes for u, v and Γ. Then, by using an adequate representation for Γ, and by exploiting the very simple form of the typing and evaluation predicate, we are able to progressively fill these holes and to close the two goals about evaluation, by repeatedly applying constructors and ad-hoc lemmas about environments.

Unlike Coq's standard `quote` mechanism, which works by conversion and has no impact on the size of proofs, this simple "user-level"-`quote` generates large

proof-terms. In fact, this is the current bottleneck of our tactic: on casual examples, the time spent in reification exceeds the time spent in computations! We thus plan to implement an efficient reification mechanism, possibly in OCaml.

2.4 Numbers, Finite Sets, and Finite Maps

To code our decision procedure, we mainly need natural numbers, finite sets, and finite maps. Coq provides several representations for natural numbers: Peano integers (`nat`), binary positive numbers (`positive`), and big natural numbers in base 2^{31} (`BigN.t`), the latter being shipped with an underlying mechanism to use machine integers and perform efficient computations. Similarly, there are various implementations of finite maps and finite sets, based on ordered lists (`FMapList`), AVL trees (`FMapAVL`), or uncompressed Patricia trees (`FMapPositive`).

While Coq's standard library features well-defined interfaces for finite sets and finite maps, the different definitions of numbers lack this standardisation. In particular, the provided tools vary greatly depending on the implementation. For example, the tactic `omega`, that decides Presburger's arithmetic on `nat`, is not available for `positive`. To abstract from this choice of basic data structures, and to obtain a modular code, we designed a small interface to package natural numbers together with the various operations we need, including sets and maps. We specified these operations with respect to `nat`, and we defined several automation tactics. In particular, by automatically translating goals to the `nat` representation, we can use the `omega` tactic in a transparent way.

3 The Algorithm and Its Proof

We now focus on the heart of our tactic: the decision procedure and the corresponding correctness proof. The algorithm that decides whether two regular expressions denote the same language can be decomposed into four steps:

1. build non-deterministic finite automata with epsilon-transitions (ϵ-NFA);
2. remove epsilon-transitions to get non-deterministic finite automata (NFA);
3. determinise the automata to obtain deterministic finite automata (DFA);
4. check that the two DFAs are equivalent.

The third step can produce automata of exponential size. Therefore, we have to carefully select our construction algorithm, so that it produces rather small automata. More generally, we have to take a particular care about efficiency; this drives our choices about both data structures and algorithms.

The Coq types we used to represent finite automata are given in Fig. 3; we use modules only for handling the namespace; the type `regex` is defined in Fig. 4, `label` and `state` are synonyms for the type of numbers. The first record type (`MAUT.t`) corresponds to the matricial representation of automata; it is rather high-level but computationally inefficient; we use it only in proofs, through the evaluation function `MAUT.eval` (`MX n m` is the type of $n \times m$ matrices over `regex`; the evaluation function calculates the matricial product $u \cdot M^\star \cdot v$ and casts it to

```
Module MAUT.
 Record t := build {
  size:    nat;
  initial: MX 1    size;
  delta:   MX size size;
  final:   MX size 1
 }.
 Definition eval(A: t): regex :=
  mx_to_scal
   ((initial A)·(delta A)*·(final A)).
End MAUT.
```

```
Module eNFA.
 Record t := build {
  size:    state;
  labels:  label;
  epsilon: state→stateset;
  delta:   label→state→stateset;
  initial: state;
  final:   state }.
 Definition to_MAUT(A: t): MAUT.t.
 Definition eval A :=
  MAUT.eval (to_MAUT A).
End eNFA.
```

```
Module NFA.
 Record t := build {
  size:    state;
  labels:  label;
  delta:   label→state→stateset;
  initial: stateset;
  final:   stateset }.
 Definition to_MAUT(A: t): MAUT.t.
 Definition eval A :=
  MAUT.eval (to_MAUT A).
End NFA.
```

```
Module DFA.
 Record t := build {
  size:    state;
  labels:  label;
  delta:   label→state→state;
  initial: state;
  final:   stateset }.
 Definition to_MAUT(A: t): MAUT.t.
 Definition eval A :=
  MAUT.eval (to_MAUT A).
End DFA.
```

Fig. 3. Coq types and evaluation functions of the four automata representations

a `regex`). The three other types are efficient representations for the three kinds of automata we mentioned above; fields `size` and `labels` respectively code for the number of states and labels, the other fields are self-explanatory. In each case, we define a translation function to matricial automata (`to_MAUT`), so that each kind of automata can eventually be evaluated into a regular expression.

The overall structure of the correctness proof is depicted in Fig. 5. Datatypes are recalled on the left-hand side; the outer part of the right-hand side corresponds to computations: starting from two regular expressions α and β, two DFAs A_3 and B_3 are constructed and tested for equivalence. The proof corresponds to the inner equalities (==): each automata construction preserves the semantics of the initial regular expressions, two DFAs evaluate to equal values when they are declared equivalent by the corresponding algorithm.

In the following sections, we give more details about each step of the decision procedure, together with a sketch of our correctness proof (although we do not implement the same algorithms, this proof is largely based on Kozen's one [20]).

3.1 Construction

There are several ways of constructing an ϵ-NFA from a regular expression. At first, we implemented Thompson's construction [33], for its simplicity; we finally switched to a variant of the Ilie and Yu's construction [17] which produces smaller automata. This algorithm constructs an automaton with a single initial state and a single accepting state (respectively denoted by i and f); it proceeds by structural induction on the given regular expression. The corresponding steps are depicted below; the first drawing corresponds to the base cases (variable, one, zero); the second one is union (plus): we recursively build the two sub-automata

```
Inductive regex :=
| dot:  regex→regex→regex
| plus: regex→regex→regex
| star: regex→regex
| one:  regex
| zero: regex
| var:  label→regex.

Inductive eq :=
| eq_trans: Transitive eq
| eq_sym: Symmetric eq
| eq_dot_zero: ∀e, e·0==0
| eq_plus_idem: ∀e, e+e==e
| ...
where "e==f" := (eq e f).
```

Fig. 4. Regular expressions

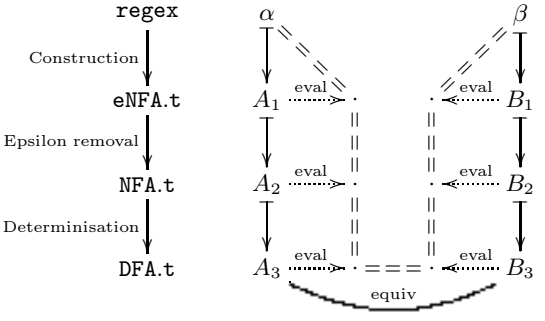
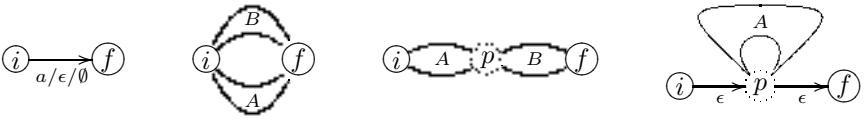


Fig. 5. Correctness of the algorithm

between i and f; the third one is concatenation: we introduce a new state, p, build the first sub-automaton between i and p, and the second one between p and f; the last one is for iteration (star): we build the sub-automata between a new state p and p itself, and we link i, p, and f with two epsilon-transitions.



To avoid costly union operations, we actually use an accumulator (an ϵ-NFA) to which we recursively add states and transitions.

We prove correctness in two steps, by using a more high-level algorithm. This algorithm is very similar to the previous one, except that it works with the matricial representation (`MAUT.t`). The following lemma states that the corresponding implementations are equivalent (`regex_to_eNFA` is the efficient function, `regex_to_MAUT` is the high-level one, and `===` is matricial automata equality):

```
Lemma step1: ∀e, eNFA.to_MAUT (regex_to_eNFA e) === regex_to_MAUT e.
```

Therefore, it suffices to prove correctness for the high-level construction—the following lemma—for which we can use algebraic and matricial reasoning.

```
Lemma step2: ∀e, MAUT.eval (regex_to_MAUT e) == e.
```

To obtain the latter lemma, we have to consider the following one, where `build` is the recursive function that underpins `regex_to_MAUT`: `build e i f A` applies the above construction to the regular expression `e`, between states `i` and `f` of the matricial automaton accumulator `A`; and `add e i f A` just adds a transition labelled `e` to `A`, between `i` and `f`.

```
Lemma step2': ∀e i f A, MAUT.eval (build e i f A) == MAUT.eval (add e i f A).
```

As an example of the kind of algebraic reasoning we need to formalise, the following property of star w.r.t. block matrices is used twice: with $(x, y, z) = (e, 0, f)$, it gives the case of a concatenation $(e \cdot f)$; with $(x, y, z) = (1, e, 1)$ it

yields iteration ($e^\star$). In both cases, the line and the column that are added on the left-hand side correspond to the state (p) generated by the construction.

$$\left[\, u \,\middle|\, 0 \,\right] \cdot \left[\begin{array}{ccc|c} & \vdots & & 0 \\ \cdots & M_{i,f} & \cdots & x \\ & \vdots & & 0 \\ \hline 0 & z & 0 & y \end{array}\right]^\star \cdot \left[\frac{v}{0}\right] = \quad u \cdot \left[\begin{array}{c} \vdots \\ \cdots\ M_{i,f} + x \cdot y^\star \cdot z \cdots \\ \vdots \end{array}\right]^\star \cdot v$$

Finally, by combining the lemmas `step1` and `step2`, we obtain the correctness of our construction algorithm, i.e., we can fill the two triangles from Fig. 5:

```
Theorem construction_correct: ∀e, eNFA.eval (regex_to_eNFA e) == e.
```

3.2 Epsilon Transitions Removal

The automata obtained with the above construction contain epsilon-transitions: their transitions matrices are of the form $M = J + N$ with $N = \sum_a a \cdot N_a$, where J and the N_a are 0-1 matrices. J and N respectively correspond to the graph of epsilon and labelled transitions. Algebraically, removing epsilon-transitions is achieved using a simple law: $\forall xy, (x + y)^\star = x^\star \cdot (y \cdot x^\star)^\star$. This law yields

$$u \cdot (J + N)^\star \cdot v = u \cdot J^\star \cdot (N \cdot J^\star)^\star \cdot v \ ,$$

so that automata $\langle u, N, v\rangle$ and $\langle u \cdot J^\star, N \cdot J^\star, v\rangle$ are equivalent. We furthermore check that the latter automaton no longer contains epsilon-transitions: this is a NFA. This algebraic proof is not surprising: looking at 0-1 matrices as binary relations, $J^\star$ actually corresponds to the reflexive-transitive closure of J.

Although this is how we prove the correctness of this step, computing $J^\star$ algebraically is inefficient: we have to implement a proper transitive closure algorithm for the low-level representation of automata. We actually rely on a property of the construction from Sect. 3.1: when given regular expressions in "strict star" form, the produced ϵ-NFAs have acyclic epsilon-transitions. More precisely, we say that a regular expression is in *strict star form* if for all its sub-expressions of the form $e^\star$, the regular language corresponding to e does not contain the empty word. Intuitively, the only possibility for introducing an epsilon-cycle in the construction from Sect. 3.1 comes from star expressions. By forbidding the empty word to appear in such cases, we prevent the formation of epsilon-cycles.

Concretely, this means that: 1) we wrote a recursive function that transforms a regular expression into an equivalent one, in strict star form; 2) we proved that our construction algorithm returns ϵ-NFAs whose reversed epsilon-transitions are well-founded, when the argument is in strict star form; 3) based on this assumption we implemented a simple transitive closure algorithm, using well-founded recursion and memoisation; 4) we proved that this algorithm actually yields an automaton (of type `NFA.t`) whose translation into a matricial automaton is exactly $\langle u \cdot J^\star, N \cdot J^\star, v\rangle$, so that the above algebraic proof applies.

3.3 Determinisation

Determinisation is exponential in worst case: this is a power-set construction. However, examples where this bound is reached are rather contrived: the empirical complexity is tractable. Starting from a NFA $\langle u, M, v\rangle$ with n states, the algorithm consists in a standard depth-first enumeration of the subsets accessible from the set of initial states. It returns a DFA $\langle \widehat{u}, \widehat{M}, \widehat{v}\rangle$ with $\widehat{n}$ states, together with a injective map ρ from $[1..\widehat{n}]$ to subsets of $[1..n]$. We sketch the algebraic part of the correctness proof. Let X be the rectangular $(\widehat{n}, n)$ 0-1 matrix defined by $X_{sj} \triangleq j \in \rho(s)$, we prove that the following commutation properties hold:

$$\widehat{M} \cdot X = X \cdot M \quad (1) \qquad \widehat{u} \cdot X = u \quad (2) \qquad \widehat{v} = v \cdot X \quad (3)$$

The intuition is that X is a "decoding" matrix: it sends states of the DFA to the characteristic vectors of the corresponding subsets of the NFA. Therefore, (1) can be read as follows: executing a transition in the DFA and then decoding the result is equivalent to decoding the starting state and executing parallel transitions in the NFA. Similarly, (2) states that the initial state of the DFA corresponds to the set of initial states of the NFA. From (1), we deduce that $(\widehat{M})^\star \cdot X = X \cdot M^\star$ using a theorem of Kleene algebras; we conclude with $(2, 3)$:

$$\widehat{u} \cdot (\widehat{M})^\star \cdot \widehat{v} = \widehat{u} \cdot (\widehat{M})^\star \cdot X \cdot v = \widehat{u} \cdot X \cdot M^\star \cdot v = u \cdot M^\star \cdot v \ .$$

A Coq-specific technical difficulty in the concrete implementation of this algorithm comes from termination: the main loop is executed at most 2^n times (there are 2^n subsets of $[1..n]$), but we cannot use this bound directly. Indeed, we can easily determinise NFAs with 500 states in practice, while computing 2^{500} is obviously out of reach (the binary representation of numbers does not help since we need to do structural "unary" recursion); we thus have to iterate lazily. We tried to use well-founded recursion; this was rather inconvenient, however, since this requires mixing some non-trivial proofs with the code. We currently use the following "pseudo-fixpoint operators", defined in continuation passing style:

```
Variables A B: Type.
Fixpoint linearfix n (f: (A → B) → A → B) (k: A → B) (a: A): B :=
  match n with O ⇒ k a | S n ⇒ f (linearfix n f k) a end.
Fixpoint powerfix n (f: (A → B) → A → B) (k: A → B) (a: A): B :=
  match n with O ⇒ k a | S n ⇒ f (powerfix n f (powerfix n f k)) a end.
```

Intuitively, `linearfix n f k` lazily approximates a potential fixpoint of the functional `f`: if a fixpoint is not reached after `n` iterations, it uses `k` to escape. The `powerfix` operator behaves similarly, except that it escapes after $2^n - 1$ iterations: we prove that `powerfix n f k a` is equal to `linearfix (`$2^n - 1$`) f k a`. Thanks to these operators, we can write the code to be executed using `powerfix`, while keeping the ability to reason about the simpler code obtained with a naive structural iteration over 2^n: both versions of the code are easily proved equivalent, using the intermediate `linearfix` characterisation.

3.4 Equivalence Checking

Two DFAs are equivalent if and only if their respective minimised DFAs are equal up-to isomorphism. While exploring all state permutations is sufficient to

obtain decidability, there is a more direct and efficient approach that does not require minimisation: one can perform an on-the-fly *simulation* check, using an almost linear algorithm by Hopcroft and Karp [1].

This algorithm proceeds as follow: it computes the disjoint union automata $\langle u, M, v\rangle$, and checks that the former initial states (i_A, i_B) are equivalent. Intuitively, two states are equivalent if they can match each other's transitions to reach equivalent states, with the constraint that no accepting state can be equivalent to a non-accepting one. Hence, the algorithm starts with a pair of states that must be equivalent—typically (i_A, i_B)—and try to recursively equate their reducts along transitions. To be almost linear, the algorithm uses a disjoint-sets data structure to compute equivalence classes. Indeed, if the pairs $\{i, j\}$ and $\{j, k\}$ have already been equated, one can skip the pair $\{i, k\}$ if encountered.

To our knowledge, there are two implementations of union-find data structures in Coq [9,25]. However, [9] is not computational, and [25] is more geared toward extraction (it uses `sig` types). Therefore, we had to re-implement and prove this data structure from scratch. Namely, we implemented disjoint-sets forests [10] with path compression and the usual "union by rank" heuristic.

Like previously, the correctness of the equivalence check is proved algebraically: we define a 0-1 matrix Y to encode the obtained equivalence relation on states, and we prove that it satisfies the following properties:

$$1 \leq Y \quad (1) \qquad Y \cdot Y \leq Y \quad (2) \qquad Y \cdot M \leq M \cdot Y \quad (3)$$

$$i_A \cdot Y = i_B \cdot Y \quad (4) \qquad Y \cdot v = v \quad (5)$$

Equations $(1, 2)$ correspond to the fact that Y encodes a reflexive and transitive relation. The remaining equations assess that Y is a simulation (3), that the initial arguments are equated (4), and that related states are either accepting or non accepting (5). This allows us to conclude using standard algebraic reasoning.

3.5 Completeness: Counter-Examples

By combining the proofs from the above sections according to Fig. 5, we obtain the correctness of the decision procedure: if `decide_kleene a b` returns `true`, then `a==b`, and thanks to the untyping theorem $(*)$ from Sect. 2.3, we deduce that `a` and `b` are equal in any typed Kleene algebra.

We also proved the converse implication, i.e., *completeness*. This basically amounts to exhibiting a counter-example in the case where the DFAs are not equivalent. From the algorithmic point of view, this is almost straightforward: it suffices to record the word that is being read in the algorithm from Sect. 3.4; when two states that should be equivalent differ by their accepting status, we know that the current word is accepted by one DFA and not by the other one. Accordingly, the `decide_kleene` function actually returns an `option (list label)` rather than a boolean, so that the counter-example can be given to the user.

From the proof point of view, to obtain the reverse implication of the equivalence we mentioned in Sect. 1, we just have to show that languages (i.e.,

predicates over list of labels, `list label→Prop`) form a Kleene algebra in which the language accepted by a DFA is exactly the language obtained with `DFA.eval`:

```
Theorem interp_DFA_eval: ∀ A: DFA.t, DFA_language A [=] interp (DFA.eval A).
```

(`DFA.eval` actually returns a regular expression which we need to interpret as a language; `[=]` is language equality, i.e., pointwise equivalence of the predicates.)

4 Conclusions

We presented a correct and complete reflexive tactic for deciding Kleene algebra equalities. This tactic belongs to a broader project whose aim is to provide algebraic tools for working with binary relations in Coq; the development can be downloaded from [5]. To our knowledge, this is the first certified efficient implementation of these algorithms and their integration as a generic tactic.

4.1 Performances

We performed intensive tests on randomly generated regular expressions. On typical use cases, the tactic runs instantaneously (except for the time spent in the reification mechanism, as explained at the end of Sect. 2.3). It runs in less than one second for expressions with 150 internal nodes and 20 variables, and less than one minute for even larger expressions (1000 internal nodes, 40 variables), that are very unlikely to appear in "human-written" proofs.

Thanks to the efficient implementation of radix-2 search trees (`PositiveMap`), we get higher performances if we use `positive` rather than `BigN.t`, despite the underlying mechanism that uses machine arithmetic—a recent feature of Coq.

4.2 Related Works

The idea of reasoning about binary relations algebraically is old [32,11]. Among others [18,34], Struth applied this idea within an interactive theorem prover [31]. He later turned to automated first-order theorem provers (ATP): Höfner and him verified facts about various relation algebras [15,16] using Prover9, a resolution/-paramodulation based ATP. Our approaches are quite different: we implemented a decision procedure for a decidable theory, whereas their proposal consists in feeding a generic automated prover with the axioms of some algebras, and to see how far the prover can go by itself. As a consequence, their methodology applies directly to a very wide class of goals and algebras, while we are restricted to the equational theory of Kleene algebras. On the other hand, our tactic always terminates, while Prover9 is unpredictable: even for very simple goals, it can diverge, find a proof immediately, or find a proof in a few minutes [16].

At the time we started this project, Briais formalised decidability of regular languages equality [6]. However, his approach is not computational, so that even straightforward identities cannot be checked by letting Coq compute.

Narboux defined a set of Coq tactics for diagrammatic proofs [27]. He works in the concrete setting of binary relations, which makes it possible to represent

more diagrams, but does not scale to other models. The level of automation is rather low: it basically reduces to a set of hints for the `auto` tactic.

Our notion of strict star form (Sect. 3.2) was inspired by the standard notion of *star normal form* [7] and the idea of *star unavoidability* [17]. To our knowledge, the facts that we can always rewrite regular expressions in such a form and that ϵ-NFAs with acyclic epsilon-transitions can be constructed in this way are new.

4.3 Directions for Future Work

Earlier failure checks. Our algorithm for checking equivalence of DFAs returns whenever two non-equivalent states are encountered. This optimisation greatly improves over minimisation-based algorithms, but we could go one step further, by checking the equivalence on-the-fly, during the determinisation phase. This way, we could abort the computation of DFAs as soon as a discrepancy is found.
KAT, Hoare logic. We plan to extend our decision procedure to *Kleene algebras with tests* (KAT), so as to provide automation to prove correctness of programs in Hoare logic [22]. A first possibility would be to encode KAT expressions into KA [23] and to use the current tactic. This encoding being potentially exponential, it is unclear whether this approach would be tractable. A more involved approach would be to use the dedicated automata construction presented in [8].

Acknowledgements. We thank Guilhem Moulin, Assia Mahboubi, Matthieu Sozeau, Bruno Barras, and Hugo Herbelin for highly stimulating discussions.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Allen, S.F., Constable, R.L., Howe, D.J., Aitken, W.E.: The semantics of reflected proof. In: LICS, pp. 95–105. IEEE Computer Society, Los Alamitos (1990)
3. Bertot, Y., Gonthier, G., Ould Biha, S., 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)
4. Blanqui, F., Coupet-Grimal, S., Delobel, W., Koprowski, A.: CoLoR: a Coq library on rewriting and termination (2006)
5. Braibant, T., Pous, D.: Coq library: ATBR, algebraic tools for working with binary relations (May 2009), `http://sardes.inrialpes.fr/~braibant/atbr/`
6. Briais, S.: Coq development: Finite automata theory (July 2008), `http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz`
7. Brüggemann-Klein, A.: Regular expressions into finite automata. TCS 120(2), 197–213 (1993)
8. Cohen, E., Kozen, D., Smith, F.: The complexity of Kleene algebra with tests, TR96-1598, CS Dpt., Cornell University (July 1996)
9. Conchon, S., Filliâtre, J.-C.: A Persistent Union-Find Data Structure. In: ACM SIGPLAN Workshop on ML, Freiburg, Germany, October 2007, pp. 37–45 (2007)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)
11. Doornbos, H., Backhouse, R., van der Woude, J.: A calculational approach to mathematical induction. TCS 179(1-2), 103–135 (1997)

12. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Urban, C. (ed.) 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. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in Coq. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 98–113. Springer, Heidelberg (2005)
15. Höfner, P., Struth, G.: Automated reasoning in Kleene algebra. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 279–294. Springer, Heidelberg (2007)
16. Höfner, P., Struth, G.: On automating the calculus of relations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 50–66. Springer, Heidelberg (2008)
17. Ilie, L., Yu, S.: Follow automata. Inf. and Comp. 186(1), 140–162 (2003)
18. Kahl, W.: Calculational relation-algebraic proofs in Isabelle/Isar. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS 2003. LNCS, vol. 3051, pp. 178–190. Springer, Heidelberg (2004)
19. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Automata Studies, pp. 3–41. Princeton University Press, Princeton (1956)
20. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inf. and Comp. 110(2), 366–390 (1994)
21. Kozen, D.: Typed Kleene algebra, TR98-1669, CS Dpt. Cornell University (1998)
22. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Log. 1(1), 60–76 (2000)
23. Kozen, D., Smith, F.: Kleene algebra with tests: Completeness and decidability. In: van Dalen, D., Bezem, M. (eds.) CSL 1996. LNCS, vol. 1258, pp. 244–259. Springer, Heidelberg (1997)
24. Krob, D.: Complete systems of B-rational identities. TCS 89(2), 207–343 (1991)
25. Leroy, X.: A formally verified compiler back-end. JAR 43(4), 363–446 (2009)
26. Meyer, A.R., Stockmeyer, L.J.: Word problems requiring exponential time. In: Proc. STOC, pp. 1–9. ACM, New York (1973)
27. Narboux, J.: Formalisation et automatisation du raisonnement géométrique en Coq. PhD thesis, Université Paris Sud (September 2006)
28. Pous, D.: Untyping typed algebraic structures and colouring proof nets of cyclic linear logic. Technical Report RR-7176, INRIA Rhône-Alpes (January 2010)
29. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development 3(2), 114–125 (1959)
30. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
31. Struth, G.: Calculating Church-Rosser proofs in Kleene algebra. In: de Swart, H. (ed.) RelMiCS 2001. LNCS, vol. 2561, pp. 276–290. Springer, Heidelberg (2002)
32. Tarski, A., Givant, S.: A Formalization of Set Theory without Variables, AMS, Providence, Rhode Island, vol. 41. Colloquium Publications (1987)
33. Thompson, K.: Regular expression search algorithm. ACM C. 11, 419–422 (1968)
34. von Oheimb, D., Gritzner, T.F.: RALL: Machine-supported proofs for relation algebra. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 380–394. Springer, Heidelberg (1997)

Fast LCF-Style Proof Reconstruction for Z3

Sascha Böhme[1] and Tjark Weber[2,⋆]

[1] Technische Universität München
boehmes@in.tum.de
[2] University of Cambridge
tw333@cam.ac.uk

Abstract. The Satisfiability Modulo Theories (SMT) solver Z3 can generate proofs of unsatisfiability. We present independent reconstruction of these proofs in the theorem provers Isabelle/HOL and HOL4 with particular focus on efficiency. Our highly optimized implementations outperform previous LCF-style proof checkers for SMT, often by orders of magnitude. Detailed performance data shows that LCF-style proof reconstruction can be faster than proof search in Z3.

1 Introduction

Interactive theorem provers like Isabelle/HOL [1] and HOL4 [2] have become invaluable tools in formal verification. They typically provide rich specification logics, which allow modelling complex systems and their behavior. Despite the merits of user guidance in proving theorems, there is a need for increased proof automation in interactive theorem provers: even proving a simple theorem can be a tedious task.

In recent years, automated theorem provers have emerged for combinations of first-order logic with various background theories, e.g., linear arithmetic, arrays, bit vectors [3]. These provers, called Satisfiability Modulo Theories (SMT) solvers, are of particular value in formal verification, where specifications and verification conditions can often be expressed as SMT formulas [4,5]. SMT solvers also have applications in model checking, constraint solving, and other areas.

Interactive theorem provers can greatly benefit from the reasoning power of SMT solvers. Proof obligations that are SMT formulas can simply be passed to the automated prover, which will solve them without further human guidance. However, almost every SMT solver is known to contain bugs [6]. When integrated naively, the SMT solver (and the integration) become part of the trusted code base: bugs could lead to inconsistent theorems in the interactive prover. For formal verification, where correctness is often paramount, this is undesirable.

The problem can be solved by requiring the SMT solver to produce proofs (of unsatisfiability), and reconstructing these proofs in the interactive prover.

⋆ The first author was supported by the German Federal Ministry of Education and Research under grant 01IS07008. The second author was supported by the British EPSRC under grant EP/F067909/1.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 179–194, 2010.

Among the proof-producing solvers is Z3 [7], a state-of-the-art SMT solver developed by Microsoft Research. In this paper, we present independent reconstruction for the proofs generated by Z3 in Isabelle/HOL and HOL4. These two popular LCF-style [8] systems are based on a relatively small trusted kernel (see Sect. 3) that provides a fixed set of simple inference rules. Z3, on the other hand, uses a number of powerful inference rules in its proofs (see Sect. 4). This makes proof reconstruction challenging.

Our LCF-style implementations of proof reconstruction (see Sect. 5) do not extend the trusted code base. Any attempt to perform an unsound inference will be caught by the underlying theorem prover's kernel. In contrast, a standalone proof checker for Z3 could be implemented much more efficiently, but would have to be trusted. For utmost reliability, the latter approach is clearly not ideal, also because Z3's proofs are relatively difficult to check: besides first-order reasoning also decision procedures for supported theories (e.g., arrays, linear arithmetic) are required. As a mixture of both worlds, a proof checker obtained by reflection could be much faster than our LCF-style implementation yet be formally verified. However, sparse documentation of Z3's proof rules would heavily complicate this approach, while our implementations just fall back to existing automated proof tools for underspecified cases. Maybe not surprisingly, the only proof checker applicable to all Z3 proofs that existed previous to our work was an older version of Z3 [9]. By using Isabelle/HOL and HOL4 as proof checkers, we in fact discovered previously unknown bugs in Z3.

Driven by the nature of proof obligations commonly seen in formal verification and related domains, we restrict ourselves to reconstruct proofs of first-order logic theorems over the theories of equality and uninterpreted functions, arrays, and linear integer and real arithmetic. In particular, we do not consider proof reconstruction for the theory of bitvectors.

Evaluation of our implementations (see Sect. 6) is performed on SMT-LIB problems [10]. This is because there is a large and diverse library of problems readily available, promising a good coverage of Z3's proof rules. The implicit assumption that such an approach yields a practically useful system for typical goals in Isabelle/HOL or HOL4 has already been confirmed by first users.

2 Related Work

Work on the integration of SMT solvers with LCF-style theorem provers is relatively sparse.

McLaughlin et al. [11] describe a combination of HOL Light and CVC Lite for quantifier-free first-order logic with equality, arrays and linear real arithmetic. Ge and Barrett [12] present the continuation of that work for CVC3, the successor of CVC Lite, supporting also quantified formulas and linear integer arithmetic. CVC Lite's and CVC3's proof rules are much more detailed than the ones used by Z3. For instance, CVC3 employs more than 50 rules for the theory of real linear arithmetic alone. Although one would expect this to allow for more precise (and hence, faster) proof reconstruction, McLaughlin et al. report that their

implementation is around six times slower than a decision procedure for linear real arithmetic implemented directly in HOL Light. For an in-depth comparison of our work with proof reconstruction for CVC3 see Sect. 6.

Fontaine et al. [13] describe an integration of the SMT solver haRVey with Isabelle/HOL. Their work is restricted to quantifier-free first-order logic with equality and uninterpreted functions. Hurlin et al. [14] extend this approach to quantified formulas. Skolemization is discussed in great detail. Unlike in our work, background theories (e.g., linear arithmetic, arrays) are not supported.

Recently, the first author [15] presented proof reconstruction for Z3 in the theorem prover Isabelle/HOL. We improve upon that work in both reconstruction speed and completeness (i.e., correct coverage of Z3's inference rules). We discuss similarities and differences in detail in Sect. 5, before comparing performance in Sect. 6.

Common to the above approaches is their relatively poor performance on larger problems. Evaluation is typically done on a few selected, hand-crafted toy examples. Only [12,15] use a significant number of SMT-LIB [10] benchmarks for demonstrating success rates—at the cost of long reconstruction run-times. This paper is the first to focus on efficiency, and consequently, the first to give solid evidence of attainable performance for LCF-style proof reconstruction.

3 LCF-Style Theorem Proving

The term LCF-style [8] describes theorem provers that are based on a small inference kernel. Theorems are implemented as an abstract data type, and the only way to construct new theorems is through a fixed set of functions (corresponding to the underlying logic's axiom schemata and inference rules) provided by this data type. This design greatly reduces the trusted code base. Proof procedures based on an LCF-style kernel cannot produce unsound theorems, as long as the implementation of the theorem data type is correct.

Traditionally, most LCF-style systems implement a natural deduction calculus. Theorems represent *sequents* $\Gamma \vdash \varphi$, where Γ is a finite set of *hypotheses*, and φ is the sequent's *conclusion*. Instead of $\emptyset \vdash \varphi$, we simply write $\vdash \varphi$.

The two incarnations of LCF-style systems that we consider here, i.e., HOL4 and Isabelle/HOL, are popular theorem provers for polymorphic higher-order logic (HOL) [2], based on the simply-typed λ-calculus. Both systems share the implementation language, namely Standard ML.

Although Isabelle/HOL and HOL4 implement the same logic, they differ in their internal data structures, and even in the primitive inference rules provided: some rules that are primitive in one system are derived (i.e., implemented as a combination of primitive rules) in the other. Therefore, optimization is challenging, and performance comparisons must be taken with a grain of salt. Highly optimized proof procedures typically show similar performance in the two systems [16].

On top of their LCF-style inference kernels, both Isabelle/HOL and HOL4 offer various automated proof procedures: notably a simplifier, which performs

term rewriting, a decision procedure for propositional logic, tableau- and resolution-based first-order provers, and decision procedures for Presburger arithmetic on integers and real algebra.

Substitution of type and term variables is a primitive inference rule in both Isabelle/HOL and HOL4. Consequently, substitution is typically much faster than (re-)proving the theorem's specific instance. General theorems (which we will call *schematic*) can, therefore, play the role of additional inference rules.

4 Z3: Language and Proof Terms

A detailed and perspicuous description of Z3's language and proof terms has been given in [9,15]. We briefly review the key features necessary to understand this work.

Z3's language is many-sorted first-order logic, based on the SMT-LIB language [10]. Basic sorts include bool, int and real. Interpreted functions include arithmetic operators ($+, -, \cdot$), Boolean connectives ($\vee, \wedge, \neg$), constants $\top$ and $\bot$, first-order quantifiers ($\forall, \exists$), the distinct predicate, and equality. It is worth noting that the connectives $\wedge$ and $\vee$ are polyadic functions in Z3, i.e., they can take an arbitrary number of arguments.

Z3's proof terms encode natural deduction proofs. The deductive system used by Z3 contains 34 axioms and inference rules. These range from simple rules like **mp** (modus ponens) to rules that abbreviate complex reasoning steps, e.g., **rewrite** for equality reasoning involving interpreted functions, or **th-lemma** for theory-specific reasoning. We discuss selected rules in more detail in Sect. 5.

Z3's proofs are directed acyclic graphs (DAGs). Each node represents application of a single axiom or inference rule. It is labeled with the name of that axiom or inference rule and the proposition to conclude. The edges of a proof graph connect conclusions with their premises. The hypotheses of sequents are not given explicitly. A designated root node concludes $\bot$.

5 Proof Reconstruction

Our work on Z3 proof reconstruction, although heavily optimized (and in large parts developed independently), shares certain features with the approach presented in [15], and more generally also with [14,16]. We thereby confirm that these solutions are of general interest and benefit, beyond a single theorem prover implementation. The similarities with related work are as follows.

Representation of Z3's language in higher-order logic is natural and direct. Basic types and interpreted functions have corresponding counterparts in the HOL implementations considered here. We translate uninterpreted functions and sorts into higher-order logic as variables and type variables, respectively. Arrays are translated as functions. Thus, array updates (store) become function updates, array lookup (select) reduces to function application, and extensionality is an axiom.

Representation of Z3's proofs in Standard ML is via a balanced tree, with lookup in $O(\log n)$, that maps node identifiers to proof nodes. The proof generated by Z3 is parsed, and a corresponding ML value is built. Proof nodes are given by a disjoint union. Initially, each node contains the information that is recorded explicitly in the Z3 proof. Once the corresponding inference step has been checked in the theorem prover, this information is replaced by the derived theorem. Thus, lemmas only need to be derived once, even if they are used multiple times in the proof. This technique was originally proposed in [16], where efficient LCF-style proof checking for SAT solvers is discussed.

Depth-first (postorder) traversal of the proof, starting from the root node, determines the order in which proof steps are reconstructed. If there are steps in the Z3 proof that do not contribute to the derivation of the final $\bot$, they are never checked. This technique was also adapted from [16].

Assumptions in the Z3 proof can be introduced by three rules: **asserted** and **goal** introduce assumptions made in the input problem (from which $\bot$ is derived eventually), while **hypothesis** introduces arbitrary local assumptions. These must be discharged by the **lemma** rule later. We use the axiom schema $\{\varphi\} \vdash \varphi$ (which is available in both Isabelle/HOL and HOL4) to introduce assumptions, thereby inserting them as hypotheses whenever they are used in the proof. At the very end of proof reconstruction, we check that only assumptions from the input problem remain as hypotheses.

Skolem functions introduced by Z3's proof rule **sk** are given hypothetical definitions in terms of Hilbert's choice operator (see [14,15] for details). This allows to replace the equisatisfiability relation that Z3 uses in its proofs, which has no direct counterpart in higher-order logic, with equivalence.

Local definitions are used by Z3 to introduce abbreviations for formulas. The relevant rules are **intro-def** and **apply-def**. We model locally defined abbreviations by hypothetical definitions, in much the same way as Skolem functions.

5.1 Reconstruction Techniques

Beyond these similarities, however, there are numerous ways in which our approach differs from previous ones. Noticeably, we have spent considerable time on profiling (see Sect. 6.4). This has prompted faster reconstruction techniques for many of Z3's inference rules. We distinguish four different techniques to model a Z3 inference rule in an LCF-style system:

1. as a single primitive inference or schematic theorem,
2. as a combination of primitive inferences and/or schematic theorems,
3. by applying an automated proof procedure,
4. as a combination of the above.

These techniques vary in implementation effort and performance. Primitive inference rules and schematic theorems are typically the preferred choice where possible (because they are easy to use, and as fast as it gets), but they are limited in applicability: the set of primitive rules provided by an LCF-style kernel is

fixed, and schematic theorems can only be applied to terms with a fixed structure that is known in advance. For instance, deriving $\vdash \varphi$ from a conjunction $\vdash \varphi \wedge \psi$ is within the realm of possibility, but a derivation of $\vdash \varphi$ from an arbitrarily nested conjunction $\vdash \ldots \wedge \varphi \wedge \ldots$ already requires a combination of primitive inferences and/or schematic theorem instantiations.

Automated proof procedures, on the other hand, work well for rapid prototyping. About one third of Z3's proof rules merely require propositional reasoning, and another third perform relatively simple first-order reasoning. These rules can, in principle, be implemented by a single application of (1) a fast decision procedure for propositional logic [16], or (2) an automated prover for first-order logic with equality [17], respectively. Even though (1) internally employs a state-of-the-art SAT solver, and (2) has fared well in various CASC competitions [18], the key disadvantage of automated proof procedures is that their performance is hard to control. We have achieved speedups of three to four orders of magnitude by replacing calls to these automated tools with specialized implementations (using combinations of primitive inferences and/or schematic theorems) that perform the *specific* reasoning steps required to model Z3's proof rules.

Table 1 gives an overview of the reconstruction techniques currently used for the different proof rules of Z3. We apply automated proof procedures only to theory-specific rules and to approximate four rules that Z3 never used in our extensive evaluation. If performance for the latter rules was important, they might as well be implemented using the second category of reconstruction techniques.

We now describe relevant optimizations in more detail. Since primitive inference rules in different theorem provers often show different performance relative to each other, there is not necessarily one (prover-independent) optimal approach. We discuss alternatives where appropriate.

Table 1. Proof rules and reconstruction techniques

Reconstruction technique	Proof rules
Primitive inference or schematic theorem	**asserted**, **commutativity**, **goal**, **hypothesis**, **iff-false**, **iff-true**, **iff**$\sim$, **mp**, **mp**$\sim$, **refl**, **symm**, **trans**, **true**
Combination of primitive inferences and/or schematic theorems	**and-elim**, **apply-def**, **def-axiom**, **elim-unused**, **intro-def**, **lemma**, **monotonicity**, **nnf-neg**, **nnf-pos**, **not-or-elim**, **quant-inst**, **quant-intro**, **sk**, **unit-resolution**
Automated proof procedures	**der**, **distributivity**, **pull-quant**, **push-quant**
Combination of the above	**rewrite**, **rewrite***, **th-lemma**

5.2 Propositional and First-Order Reasoning

Nested conjunctions. A recurring task in Z3's proofs is to establish equivalence of two (arbitrarily parenthesized) conjunctions $\varphi \equiv p_1 \wedge \cdots \wedge p_n$ and $\psi \equiv q_1 \wedge \cdots \wedge q_n$, where $\{p_i \mid 1 \leq i \leq n\} = \{q_i \mid 1 \leq i \leq n\}$. Such permutated conjunctions arise for two reasons. First, conjunction in Z3 is polyadic, i.e., it can take an arbitrary

number of arguments, while in Isabelle/HOL and HOL4, conjunction is a binary (right-associative) operator. For instance, $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ in Isabelle/HOL is really short for $\varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$. Second, unfolding of the distinct predicate leads to conjoined inequalities: e.g., $\mathsf{distinct}\ [x, y, z] \equiv x \neq y \wedge x \neq z \wedge y \neq z$.

A rewriting-based approach (using associativity, commutativity and idempotence of conjunction) turns out to be far too slow. Instead, we perform the required re-ordering not at the term level, but using ML data structures. We first derive theorems $\{\varphi\} \vdash p_i$ (for each p_i) by assuming φ and recursively applying conjunction elimination. These intermediate theorems are stored in a balanced tree, indexed by their conclusion. From them, we derive $\{\varphi\} \vdash \psi$ by recursion over the structure of ψ, using conjunction introduction. In a similar way, we get $\{\psi\} \vdash \varphi$. Combining both theorems, we obtain $\vdash \varphi \Leftrightarrow \psi$.

This implementation has complexity $O(n \log n)$. It improves over an implementation with quadratic complexity that had been part of the HOL4 theorem prover since 1991 [19].

Nested disjunctions are treated dual to nested conjunctions, but our implementations deviate from each other due to differences in the primitive inference rules available.

In HOL4, we first show that $\psi \equiv q_1 \vee \cdots \vee q_n$ follows from each of its disjuncts. Then we recurse over the structure of the premise $\varphi \equiv p_1 \vee \cdots \vee p_n$, using disjunction elimination to show that since each disjunct p_i implies ψ, we have $\{\varphi\} \vdash \psi$. Deriving ψ from each of its disjuncts is not completely straightforward. We achieve complexity $O(n \log n)$ by assuming ψ (thereby obtaining $\{\psi\} \vdash \psi$), and then recursively deriving $\{\vartheta_1\} \vdash \psi$ and $\{\vartheta_2\} \vdash \psi$ from $\{\vartheta_1 \vee \vartheta_2\} \vdash \psi$. This inference step is not provided as a primitive rule by the HOL4 kernel. We found an implementation that uses a combination of primitive rules to be roughly twice as fast as one that instantiates a schematic theorem $\vdash (\vartheta_1 \vee \vartheta_2 \Rightarrow \psi) \Rightarrow \vartheta_1 \Rightarrow \psi$ (and a similar theorem for ϑ_2).

In Isabelle/HOL, we show equivalence of φ and ψ by contraposition, i.e., by showing that $\neg\varphi$ and $\neg\psi$ are equivalent. This can be done in analogy to the case of conjunctions, only that instead of conjunction elimination and conjunction introduction, dual theorems for negated disjunctions must be applied. The complexity of this approach is again $O(n \log n)$.

Unit resolution implements the following inference rule, which strengthens a disjunction by removing disjuncts that have been disproved:

$$\frac{\Gamma \vdash \bigvee_{i \in I} \varphi_i \quad \langle \Gamma_i \vdash \neg\varphi_i \rangle_{i \in J}}{\Gamma \cup \bigcup_{i \in J} \Gamma_i \vdash \bigvee_{i \in I \setminus J} \varphi_i}\ \textbf{unit-resolution}$$

We model this inference rule in HOL4 by extending the technique for nested disjunctions described previously. For $i \in I \setminus J$, deriving the conclusion $\bigvee_{i \in I \setminus J} \varphi_i$ from each φ_i is done exactly as before. For $i \in J$, on the other hand, we use the fact that φ_i has been disproved, and that anything (in particular, the desired conclusion) follows from $\bot$. We found this implementation to be about 30% faster in HOL4 than the approach detailed in [16], which is more efficient only when proofs contain many successive resolution steps.

In Isabelle/HOL, we again use contraposition to model unit resolution. Assume that $\neg \bigvee_{i \in I \setminus J} \varphi_i$ holds and show that this together with the facts $\neg\varphi_i$ (for $i \in J$) implies $\neg \bigvee_{i \in I} \varphi_i$. Hence the premise $\bigvee_{i \in I} \varphi_i$ implies the conclusion $\bigvee_{i \in I \setminus J} \varphi_i$. The main step of this deduction employs the contraposition-based technique for nested disjunctions described previously.

Literal memoization. The Z3 proof rule **and-elim** deduces a conjunct from a polyadic conjunction. In an LCF-style system where conjunction is binary, this amounts to repeated application of conjunction elimination. Since **and-elim** is commonly applied to the same premise $\vdash \varphi$ several times, deducing a different conjunct each time, it is more efficient to explode $\vdash \varphi$ once and for all instead of repeatedly extracting a single conjunct. The resulting conjuncts are stored in the proof node that derived $\vdash \varphi$, indexed by a balanced tree for efficient lookup. This memoization technique applies dually to the rule **not-or-elim**.

Quantifier instantiations in Z3's proof rules can be determined by (first-order) term matching. No first-order proof search is necessary. We avoid the automated first-order provers that are built into Isabelle/HOL and HOL4: they are unnecessarily powerful, but relatively slow. Instead, we perform the required combinations of primitive inferences directly.

5.3 Theory-Specific Reasoning

With these optimizations in place, Z3's propositional and first-order inference steps are checked with reasonable efficiency. The one remaining performance hog is theory-specific reasoning, involving interpreted functions (e.g., linear arithmetic). This is performed by three proof rules in Z3: **rewrite**, **rewrite***[1] and **th-lemma**. We implement these rules by sequentially trying schematic theorems, exploiting associativity, commutativity and idempotence of conjunction and disjunction, trying the simplifier, and applying decision procedures for linear integer and real arithmetic.

When done naively, the automated tools, i.e., the simplifier and the decision procedures for linear arithmetic, dominate run-time. We were able to improve performance by reducing the number of proof obligations that are passed to these tools. In our current implementation the simplifier only rewrites array updates, but not Boolean or arithmetic operators: these are handled through schematic theorems or specialized proof procedures. We employ the following optimization techniques for theory-specific reasoning.

Schematic theorems. Matching a theorem's conclusion against a given term and, if successful, instantiating the theorem accordingly is typically much faster than deriving the instance again. By studying the actual usage of **rewrite** in Z3's proofs, we identified more than 230 useful schematic theorems. These include propositional tautologies such as $\vdash (p \Rightarrow q) \Leftrightarrow (q \vee \neg p)$, theorems about equality, e.g., $\vdash (x = y) \Leftrightarrow (y = x)$, and theorems of linear integer and real arithmetic,

[1] Since **rewrite*** is a variant of **rewrite**, we implicitly include the former when referring to the latter in the remainder of this section.

e.g., $\vdash x + 0 = x$. Together, these theorems allow about 76% of all terms given to **rewrite** to be proved by instantiation alone. Because of their generality, our schematic theorems should be useful for a wide range of benchmarks. We store all schematic theorems in a term net to allow faster search for a match.

To a smaller extent, we also use schematic theorems in the implementations of Z3's proof rules **def-axiom** and **th-lemma**.

Theorem memoization. Isabelle/HOL and HOL4 allow instantiating free variables in a theorem, while Z3 has to re-derive theorems that differ in their uninterpreted functions. Hence, there is more potential for theorem re-use in these provers than in Z3. We exploit this by storing theorems of linear arithmetic that are proved by **rewrite** or **th-lemma** in a term net. Since every theorem is also stored in a proof node anyway, this increases memory requirements only slightly (namely by the memory required for the net's indexing structure). Before invoking a decision procedure for linear arithmetic on a proof obligation, we attempt to retrieve a matching theorem from the net. However, proof obligations that occur frequently are often available as schematic theorems already. Therefore, with an extensive list of schematic theorems in place, the performance gained by theorem memoization is relatively small.

Generalization. We generalize proof obligations by replacing sub-terms that are outside the fragment of linear arithmetic with variables, before passing the proof obligation to the arithmetic decision procedures. This has two benefits. First, it makes theorem memoization more useful, since more general theorems potentially can be re-used more often. Second, it avoids expensive preprocessing inside the arithmetic decision procedures. For instance, HOL4's arithmetic decision procedures perform case splitting of if-then-else expressions. This could lead to an exponential number of cases. Z3's proof rule **th-lemma**, however, does not require the linear arithmetic reasoner to know about if-then-else: if necessary, conditionals are split using one of the other proof rules before Z3 solves the problem by linear arithmetic. Therefore, proof obligations are provable even with all conditionals treated as atomic.

6 Experimental Results

We evaluated our implementations in four ways. First, we measured success rates and run-times of proof reconstruction for 1273 SMT-LIB benchmarks drawn from the latest SMT-COMP [20], an annual competition of SMT solvers. Second, a selection of these benchmarks was taken to compare our implementations with proof reconstruction for CVC3 in HOL Light [11,12] (CH). Third, we contrasted our work with previous work on proof reconstruction for Z3 [15] (ZI). Finally, we measured profiling data to give a deeper insight into our results.

Evaluation was performed on problems comprising first-order formulas (partly quantifier-free, QF, partly with (+p) or without (-p) quantifier patterns) over (combinations of) the theories of equality and uninterpreted functions (UF), arrays (A), linear integer arithmetic (LIA), linear real arithmetic (LRA), combined linear arithmetic (LIRA), integer difference logic (IDL), real difference

logic (RDL). SMT-LIB logic names are formed by concatenation of the theory abbreviations given in parentheses.

We obtained all figures[2] on a Linux system with an Intel Core2 Duo T7700 processor, running at 2.4 GHz—the same machine that had been used to evaluate ZI. Measurements were conducted with Z3 2.3 and CVC3 2.2. As underlying ML environment, we used Poly/ML 5.3.0 for both Isabelle/HOL and HOL4. For comparability with ZI, we restricted proof search to two minutes and proof reconstruction to five minutes, and limited memory usage for both steps to 4 GB. All measured times are CPU times (with garbage collection in Poly/ML excluded).

Run-times for Isabelle/HOL are typically within a factor of 1–2 of HOL4 run-times. This is because we have fully implemented some of the optimizations described in this paper only for HOL4 so far. It should not be taken as an indication that HOL4 is more efficient than Isabelle/HOL per se.

6.1 SMT-COMP Benchmarks

Table 2 shows our results for Isabelle/HOL. For every SMT-LIB logic, we measured for Z3 the average time to find a proof and the average proof size, and for our implementation the average time to reconstruct a proof (timeouts are counted as 300 s). Additionally, we give success and timeout rates for proof reconstruction and the ratio of reconstruction time to solving time (*R-time*).

Our reconstruction succeeds on 75% of all problems solved by Z3. Failures are mostly due to timeouts (19%), but also due to shortcomings of Z3 and in a few cases of our reconstruction.[3] Note that low success rates, which are in most cases caused by timeouts, occur mainly in logics dominated by arithmetic. Closer analysis (of individual examples and profiling data, see Sect. 6.4) suggests that the theory-specific proof rules **rewrite** and **th-lemma** are to blame for this deficiency.

Proofs produced by Z3 may be extremely large. Our implementations are nevertheless able to reconstruct huge proofs within the given timeout. The largest proof successfully reconstructed had a size of 168 MB and comprised more than 3 million Z3 proof rules.

Reconstruction for individual logics is at least 2.7 times slower than proof search; on average the ratio lies at 18.5 despite our extensive optimizations. A thorough study of our measurements, however, reveals that for several problem classes, the picture is different: e.g., in case of AUFLIA−p, AUFLIA+p and AUFLIRA, our reconstruction is faster than Z3 on 11-34% of all problems.

6.2 Comparison with CH

In the implementation of CH we tested, proof reconstruction was tuned for logics including uninterpreted functions, arrays and linear integer arithmetic. Thus, we only compare relevant results of the previous section with CH. Table 3 shows

[2] Our data is available at `http://www4.in.tum.de/~boehmes/fast_proof_rec.html`

[3] Z3 discovers injectivity of functions and uses this property for rewriting; reconstruction would require yet another special case for the already complex **rewrite** rule, which we have not implemented so far.

Table 2. Experimental results (Isabelle/HOL) for selected SMT-COMP logics

Logic	Solved (Z3)			Reconstructed		Rates		
	#	Time	Size	#	Time	Success	Timeout	R-time
AUFLIA+p	187	0.095 s	64 KB	187	0.413 s	100%	0%	4.34
AUFLIA−p	192	0.117 s	81 KB	190	1.962 s	98%	0%	16.72
AUFLIRA	189	0.292 s	366 KB	144	0.794 s	76%	0%	2.72
QF_AUFLIA	92	0.158 s	694 KB	49	136.498 s	53%	42%	863.85
QF_IDL	40	2.322 s	12 MB	19	173.875 s	47%	52%	74.89
QF_LIA	100	17.154 s	77 MB	26	208.713 s	26%	65%	12.17
QF_LRA	88	4.849 s	10 MB	55	142.351 s	62%	36%	29.36
QF_RDL	52	9.773 s	16 MB	26	173.953 s	50%	50%	17.80
QF_UF	87	16.131 s	62 MB	73	73.242 s	83%	16%	4.54
QF_UFIDL	55	4.511 s	12 MB	8	260.351 s	14%	85%	57.72
QF_UFLIA	91	1.543 s	4 MB	85	29.086 s	93%	6%	18.85
QF_UFLRA	100	0.086 s	914 KB	100	3.916 s	100%	0%	45.68
Total	1273	3.656 s	13 MB	962	67.785 s	75%	19%	18.54

Table 3. Comparison between Z3/Isabelle/HOL and CVC3/HOL Light

Logic	Solved			Reconstructed			
	#	Time		Rate		Time	
		Z3	CVC3	I	H	I	H
AUFLIA+p	182	0.043 s	0.485 s	100%	84%	0.294 s	12.369 s
AUFLIA−p	173	0.050 s	0.149 s	99%	80%	0.165 s	5.791 s
QF_AUFLIA	89	0.053 s	3.150 s	52%	68%	17.197 s	4.068 s
QF_IDL	34	0.483 s	10.063 s	52%	41%	20.776 s	87.772 s
QF_LIA	18	0.398 s	25.587 s	55%	0%	33.870 s	n/a
QF_UF	58	1.531 s	7.920 s	100%	86%	13.465 s	14.645 s
QF_UFIDL	38	0.301 s	6.525 s	18%	18%	13.017 s	27.120 s
QF_UFLIA	82	0.034 s	4.114 s	100%	9%	4.897 s	18.866 s
Total	674	0.219 s	3.326 s	85%	64%	4.994 s	12.147 s

Table 4. Experimental results (HOL4) for selected SMT-LIB logics

Logic	Solved (Z3)		Reconstructed			Failed		Ratio
	#	Time	#	Time	Size	#T	#Z	R-time
AUFLIA	100	0.180 s	100	0.450 s	206 KB	0	0	2.5
AUFLIRA	100	0.051 s	97	0.034 s	16 KB	0	3	0.7
QF_UF	96	2.992 s	74	16.199 s	16 MB	1	21	5.4
QF_UFLIA	99	0.534 s	92	6.948 s	194 KB	7	0	13.0
QF_UFLRA	100	0.189 s	100	1.705 s	1 MB	0	0	9.0

for each considered logic the number of problems solved by both Z3 and CVC3, their average run-time, the success rate of reconstruction for Isabelle/HOL (I) and HOL Light (H), and the average run-time of reconstruction (timeouts are counted as 300 s). Measuring these figures stimulated improvements to the implementation of CH; we only give the newest (and best) results for CH.

Clearly, our implementations can reconstruct more problems. More importantly, our reconstruction is on average more than two times faster than CH (and up to 42 times faster in the case of AUFLIA+p), even though CH does not have any parsing overhead (it uses a binary interface to CVC3, not a file-based one like our implementations).

6.3 Comparison with ZI

For comparability, we evaluated our implementations on the same set of SMT-LIB benchmarks (and, in fact, on the very same Z3 proofs) that were used to evaluate ZI [15]. Table 4 summarizes our experimental results for HOL4. For each SMT-LIB logic, the table shows the number of problems that Z3 determined to be unsatisfiable, the average run-time of Z3 (as reported in [15]), the number of successful proof reconstructions along with average HOL4 run-time and average Z3 proof size, and the number of failed proof reconstructions (due to timeouts #T and confirmed (and by now fixed) Z3 bugs #Z). The rightmost column of Tab. 4 shows the ratio of reconstruction time to solving time (R-time, cf. Sect. 6.1). We observe that this ratio is less than 1 for the AUFLIRA logic: LCF-style proof reconstruction for this logic is faster than proof search in Z3.

A more detailed comparison revealed that our highly optimized implementations (both in Isabelle/HOL and HOL4) outperform ZI on every problem of the AUFLIA, AUFLIRA, QF_UF and QF_UFLRA logics. Often, the performance gain is several orders of magnitude. Only the QF_UFLIA logic contains 18 problems for which reconstruction in HOL4, despite our optimizations, is slower than reported in [15]. We conclude that there is still potential for optimization in HOL4's decision procedure for integer arithmetic [21].

The figure to the right shows the average speedups of Isabelle/HOL (left) and HOL4 (right) over the run-times measured in [15]. Shaded bars give the maximum speedups achieved on individual problems (3,437 in the case of AUFLIRA for our HOL4 implementation!). Note that the figure uses a logarithmic scale. The overall speedup is 13.9 for HOL4 and 11.7 for Isabelle/HOL.

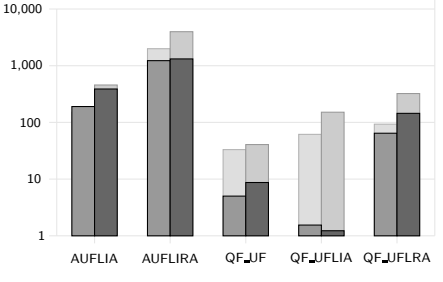


Fig. 1. Speedup factors

6.4 Profiling

To further understand these results and to identify potential for future optimization, we present relevant profiling data for our HOL4 implementation. (Isabelle/HOL profiling data is roughly similar.) Figures 2 to 6 show bar graphs that indicate the four[4] most time-consuming proof rules of Z3 for the respective SMT-LIB logic, and their percentaged shares of total run-time (dark bars).

[4] For QF_UFLIA, we only show **th-lemma** separately and combine all other rules.

Additionally, time spent on parsing proof files is shown as well (see Tab. 4 for average proof sizes). We contrast each proof rule's relative run-time with the mean frequency of that rule (light bars).

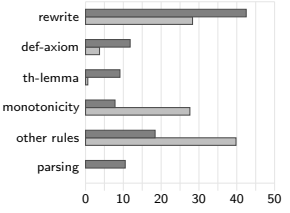


Fig. 2. AUFLIA

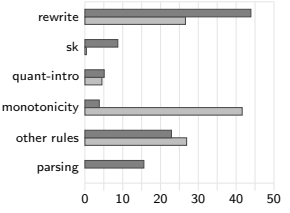


Fig. 3. AUFLIRA

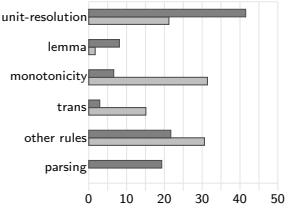


Fig. 4. QF_UF

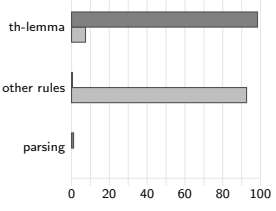


Fig. 5. QF_UFLIA

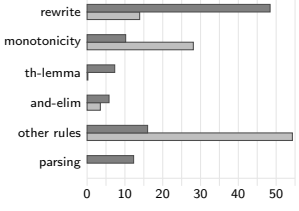


Fig. 6. QF_UFLRA

We see that after extensive optimization, proof reconstruction times are relatively well-balanced between Z3's different proof rules for most logics, although the **rewrite** rule still accounts for almost half of the total run-time on the AUFLIA, AUFLIRA, and QF_UFLRA benchmarks. For QF_UF, on the other hand, rewriting is relatively unimportant, but proofs contain many (in fact, over 5 million) **unit-resolution** inferences. Checking these consequently requires more than 41% of the run-time.

For these four logics, merely parsing Z3's proof files accounts for 11% (AUFLIA) to 19% (QF_UF) of the total run-time. Note that parsing does not involve the LCF-style inference kernel. Hence, there are limits to future performance gains that can be achieved through further optimization of LCF-style reasoning.

The picture looks different for the QF_UFLIA logic, where run-time is dominated almost entirely by the **th-lemma** rule. Parsing and other proof rules of Z3 account for less than 2% of reconstruction time. Z3 internally uses the Simplex algorithm to decide linear arithmetic [9] and applies a branch and cut strategy for integers [22]. However, any information about how a decision is found is kept private: the **th-lemma** rule only represents the statement that a system of linear inequations is inconsistent. Consequently, reconstructing this proof rule amounts to finding the refutation again, with (probably) far less optimized decision procedures in the case of Isabelle/HOL and HOL4. Instead of making those faster, we conjecture that enriching **th-lemma** with the necessary information (which is already available in Z3) would improve efficiency of proof reconstruction considerably.

7 Conclusions

We have presented LCF-style proof reconstruction for Z3 in the theorem provers Isabelle/HOL and HOL4. In comparison to a recent implementation in Isabelle/HOL [15], our implementations are significantly faster on most benchmarks, often by orders of magnitude. Moreover, we have modeled the proof rules of Z3 in Isabelle/HOL and HOL4 with unprecedented accuracy, thereby achieving almost full (except for very few exotic corner cases) proof coverage. We also outperform a related implementation of proof reconstruction for CVC3 in HOL Light [12]. We have three main conclusions.

LCF-style proof checking for SMT is feasible. Our implementations give evidence that LCF-style proof checking for SMT solvers is not only possible in principle, but also that it is feasible. Clearly there is a steep price (in terms of performance) that one has to pay for checking proofs in a general-purpose LCF-style theorem prover. However, even for proofs with millions of inferences, LCF-style proof checking can be as fast as (or even faster than) proof search in Z3. This confirms a similar observation made in [16] regarding the feasibility of LCF-style proof checking for large SAT-solver generated proofs.

Specialized implementations can be significantly faster than generic proof procedures in LCF-style provers. The speedup that we achieved over [15] shows the importance of profiling when performance is an issue. We achieved speedups of several orders of magnitude by replacing calls to generic automated proof procedures with specialized implementations that perform the specific inferences required to check Z3's proof rules. This is despite the fact that some of these automated procedures employ state-of-the-art algorithms internally. Of course, writing fast specialized proof procedures requires much more familiarity with the theorem prover than simply calling automated proof procedures.

Z3's proof format could be easier to check. Conceptually, we only had to overcome minor hurdles to implement proof reconstruction for Z3's natural-deduction style proofs in the considered LCF-style theorem provers. That the conclusion of each inference step is given explicitly proved tremendously helpful. Proof rules **rewrite** and **th-lemma**, however, seem overly complex, and despite substantial optimization efforts, they still dominate run-time in our implementations. We encourage the Z3 authors to (1) replace **rewrite** by a collection of simpler rules with clear semantics and less reconstruction effort, ideally covering specific rewriting steps of at most one theory, and (2) enrich **th-lemma** with additional easily-checkable certificates or trace information guiding refutations to avoid invocations of expensive (arithmetic) decision procedures. Currently the theoretical complexity of proof checking for background theories is the same as for proof search. We also hope that our experience will influence the design of a future SMT proof standard.

We have integrated proof reconstruction as an automated proof procedure in both HOL4 and Isabelle/HOL. If φ is an SMT formula, the user can invoke this proof procedure to have it pass $\neg\varphi$ to Z3, reconstruct Z3's proof of

unsatisfiability (if one is found) to obtain $\{\neg\varphi\} \vdash \bot$, and from this $\vdash \varphi$ is derived by contradiction. Our implementations are freely available[5] and already in use.

There are numerous differences in internal data structures between HOL4, Isabelle/HOL and other LCF-style theorem provers, but based on previous experience [16] we have little doubt that the optimization techniques presented in this paper can be used to achieve similar performance in other theorem provers.

Future work includes (1) proof reconstruction for other SMT-LIB theories, e.g., bit vectors, (2) evaluation of proof reconstruction for typical goals of Isabelle/HOL or HOL4, (3) parallel proof reconstruction [23], by checking independent paths in the proof DAG concurrently, and (4) investigations into proof compression [24] for SMT proofs.

Acknowledgments. The authors are grateful to Nikolaj Bjørner and Leonardo de Moura for their help with Z3, to Yeting Ge for his help on proof reconstruction for CVC3, and to Alexander Krauss and Lukas Bulwahn for commenting on an earlier draft of this paper. Additionally, the second author would like to thank Hasan Amjad and Mike Gordon for their support.

References

1. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
2. Gordon, M.J.C., Pitts, A.M.: The HOL logic and system. In: Towards Verified Systems. Real-Time Safety Critical Systems Series, vol. 2, pp. 49–70. Elsevier, Amsterdam (1994)
3. Kroening, D., Strichman, O.: Decision Procedures – An Algorithmic Point of View. Springer, Heidelberg (2008)
4. Collavizza, H., Gordon, M.: Integration of theorem-proving and constraint programming for software verification. Technical report, Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (2008)
5. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie — An Interactive Prover-Backend for the Verifying C Compiler. J. Automated Reasoning 44(1-2), 111–144 (2010)
6. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: 7th International Workshop on Satisfiability Modulo Theories, SMT '09 (2009)
7. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
9. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, CEUR Workshop Proceedings. vol. 418, CEUR-WS.org (2008)
10. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2 (August 2006), `http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf` (retrieved January 21, 2010)

[5] See `http://hol.sourceforge.net` and `http://isabelle.in.tum.de`

11. McLaughlin, S., Barrett, C., Ge, Y.: Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. Electronic Notes in Theoretical Computer Science 144(2), 43–51 (2006)
12. Ge, Y., Barrett, C.: Proof translation and SMT-LIB benchmark certification: A preliminary report. In: 6th International Workshop on Satisfiability Modulo Theories, SMT '08 (2008)
13. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
14. Hurlin, C., Chaieb, A., Fontaine, P., Merz, S., Weber, T.: Practical proof reconstruction for first-order logic and set-theoretical constructions. In: Proceedings of the Isabelle Workshop 2007, Bremen, Germany, July 2007, pp. 2–13 (2007)
15. Böhme, S.: Proof reconstruction for Z3 in Isabelle/HOL. In: 7th International Workshop on Satisfiability Modulo Theories, SMT '09 (2009)
16. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. J. Applied Logic 7(1), 26–40 (2009)
17. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Design and Application of Strategies/Tactics in Higher Order Logics (STRATA '03), pp. 56–68 (2003); Number NASA/CP-2003-212448 in NASA Technical Reports
18. Hurd, J.: Metis performance benchmarks, `http://www.gilith.com/software/metis/performance.html` (retrieved January 21, 2010)
19. HOL88 contributors: HOL88 source code, `http://www.ftp.cl.cam.ac.uk/ftp/hvg/hol88/holsys.tar.gz` (retrieved January 21, 2010)
20. Barrett, C., Deters, M., Oliveras, A., Stump, A.: 5th Annual Satisfiability Modulo Theories Competition. In: SMT-COMP '09 (2009), `http://www.smtcomp.org/2009/`
21. Norrish, M.: Complete integer decision procedures as derived rules in HOL. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 71–86. Springer, Heidelberg (2003)
22. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
23. Wenzel, M.: Parallel proof checking in Isabelle/Isar. In: ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (2009)
24. Amjad, H.: Data compression for proof replay. J. Automated Reasoning 41(3-4), 193–218 (2008)

The Optimal Fixed Point Combinator

Arthur Charguéraud

INRIA
arthur.chargueraud@inria.fr

Abstract. In this paper, we develop a general theory of fixed point combinators, in higher-order logic equipped with Hilbert's epsilon operator. This combinator allows for a direct and effective formalization of corecursive values, recursive and corecursive functions, as well as functions mixing recursion and corecursion. It supports higher-order recursion, nested recursion, and offers a proper treatment of partial functions in the sense that domains need not be hardwired in the definition of functionals. Our work, which has been entirely implemented in Coq, unifies and generalizes existing results on contraction conditions and complete ordered families of equivalences, and relies on the theory of optimal fixed points for the treatment of partial functions. It provides a practical way to formalize circular definitions in higher-order logic.

1 Introduction

1.1 Motivation: Partial Corecursive Functions

To the best of our knowledge, there exists, until now, no general approach to formalizing partial corecursive functions in a simple and satisfying manner. Consider for example the filter function on infinite streams. Given a predicate P of type $A \to \mathsf{bool}$ (or $A \to \mathsf{Prop}$), the filter function f takes a stream s and returns a stream made of the elements of s that satisfy P. The filter function is partial because it produces a well-defined stream only when its argument s contains infinitely many items satisfying the predicate P.

One way to constructively formalize the definition of filter in a logic of total functions is to have f take as extra argument a proof that its argument contains infinitely many items satisfying the predicate P. In this approach, studied by Bertot [4], the new filter function does not have the type $\mathsf{stream}\, A \to \mathsf{stream}\, A$, but instead admits a dependent type. Unfortunately, working with dependent types is often associated with numerous technical difficulties, so we would rather find a solution that does not make such a heavy use of dependent types.

A different, non-constructive approach to formalizing the filter function was proposed by Matthews [17]. To apply his technique, the filter function first needs to be turned into a total function, by testing explicitly whether the argument belongs to the domain. Let "$\mathsf{never}\, P\, s$" be a predicate that holds when the stream s does not contain any item satisfying P. The body of the filter function can be described through a functional F, as follows. Throughout the paper, the operator :: denotes the consing operation on infinite streams.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 195–210, 2010.

$$F\ f\ s \quad \triangleq \quad \mathsf{let}\ x :: s' = s\ \mathsf{in}\ \mathsf{if}\ (\mathsf{never}\ P\ s)\ \mathsf{then}\ \mathsf{arbitrary}\ \mathsf{else}\ \mathsf{if}\ (P\ x)\ \mathsf{then}\ x :: (f\ s')\ \mathsf{else}\ f\ s'$$

The filter function f can then be defined as "$\mathsf{Fix}_1\ F$", where Fix_1 is a combinator that picks, using Hilbert's epsilon operator, the unique fixed point of its argument when it exists, and otherwise returns an arbitrary value. Here, the functional F can be proved to admit a unique fixed point using a fixed point theorem based on *contraction conditions*, devised by Matthews [17]. It follows that f satisfies the fixed point equation $f\ s = F\ f\ s$ for any stream s.

The main downside of the approach described above is that the domain of the function needs to be hardwired in its definition. As argued by Krauss [12] for the case of recursive functions, this requirement is unsatisfactory. First, it requires us to modify the code of the functional, which is inelegant and may cause difficulties when extracting executable code. Second, it overspecifies the output of the function outside its domain. Third, it requires knowledge of the domain of the function at the time of its definition, which is not always practical [12].

The central matter of this paper is to construct a fixed point combinator Fix that truly supports partial functions. For example, Fix can be directly applied to the functional that describes the original filter function, shown below.

$$F\ f\ s \quad \triangleq \quad \mathsf{let}\ x :: s' = s\ \mathsf{in}\ \mathsf{if}\ (P\ x)\ \mathsf{then}\ x :: (f\ s')\ \mathsf{else}\ f\ s'$$

1.2 Fixed Point Equations with Non-unique Solutions

Most forms of circular definitions can be captured by (or encoded as) an equation of the form $a = F\ a$. Yet, such a fixed point equation does not necessarily admit a unique solution.

One typical case is that of partial functions. In a logic of total functions, a partial function can be represented as a pair of a total function f of type $A \to B$ and a domain D of type $A \to \mathsf{Prop}$. The partial function (f, D) is said to be a partial fixed point of a functional F of type $(A \to B) \to (A \to B)$ if the equation $f\ x = F\ f\ x$ holds for any x satisfying D. (We postpone to §3.3 the discussion of how circular definitions for partial functions can be expressed as equations of the form $a = F\ a$.) A functional F typically admits several partial functions as fixed point. Can one of them be considered the "best" fixed point for F?

The starting point of this paper is the observation that the exact answer to this question is given by the theory of the *optimal fixed point* developed in 1975 by Manna and Shamir [16], which we have formalized in Coq. A fundamental idea in this theory is that the only genuine solutions of a fixed point equation are the partial functions that are consistent with any other fixed point (two functions are consistent if they agree on the intersection of their domain). Such fixed points are said to be *generally-consistent*. The optimal fixed point is defined as the generally-consistent fixed point with the largest domain. In a sense, the optimal fixed point is the most well-defined solution that can be extracted from the fixed point equation without making arbitrary choices. Manna and Shamir [16] proved that any functional of type $(A \to B) \to (A \to B)$ admits an optimal fixed point.

Another typical case where fixed point equations do not not admit unique solutions occurs when working modulo equivalence relations. A value a is a fixed point of F modulo $(\equiv)$ if the equation $x \equiv F\,x$ holds for any value x such that $x \equiv a$. Moreover, a fixed point a is said to be the unique fixed point modulo $(\equiv)$ of a functional F if any other fixed point x of F is equivalent to a modulo $(\equiv)$. In this case, even though the functional F does not admit a unique solution, it admits a unique equivalence class of solutions. Thus, any element from this class of equivalence can be considered as representing the meaning of the the circular definition associated with F. The interest of the definition of "fixed point modulo" is that it allows defining recursive functions on values compared modulo an equivalence relation without involving an explicit quotient structure.

1.3 A Generic Fixed Point Combinator

In order to unify the various forms of circular definitions, we introduce a generic fixed point combinator. The basic idea is to pick the "best" fixed point, for a customizable notion of "best" that depends on the kind of circular definition being targeted. Our combinator, called Fix, takes as argument an equivalence relation $\equiv$, an order relation $\lhd$ and a functional F. It then uses Hilbert's epsilon operator to pick, among the set of all fixed points modulo $\equiv$ of the functional F, the greatest fixed point with respect to $\lhd$ (not be confused with the greatest fixed point in the sense of domain theory).

$$\mathsf{Fix}\,(\equiv)\,(\lhd)\,F \quad \triangleq \quad \epsilon\,x.\,[\,\mathsf{greatest}\,(\lhd)\,(\mathsf{fixed\_point\_modulo}\,(\equiv)\,F)\,x\,]$$

Appropriate instantiations of the binary relations $\equiv$ and $\lhd$ produce a combinator for unique fixed point and a combinator for optimal fixed point (possibly modulo an equivalence relation).

Now, in order to exploit properties about the value returned by $\mathsf{Fix}\,(\equiv)\,(\lhd)\,F$, we need to prove that the functional F indeed admits a greatest fixed point. For a very large scope of circular definitions, the existence of greatest fixed points can be derived from one very general theorem, which is developed in this paper. This theorem combines and generalizes several existing ideas: *contraction conditions* [11], *inductive invariants* [14] and *complete ordered families of equivalence* [17,9]. Moreover, the corollaries used in the particular case of partial functions rely on the theory of *optimal fixed points* [16] and involves a generalized version of a theorem developed in the context of *maximal inductive fixed points* [14,13].

The paper is organized as follows. First, we present all the ingredients that our paper builds upon. Second, we describe our generic fixed point combinator and its specialized versions. We then present our fixed point theorem and its corollaries. Finally, we investigate, without formal justification, the possibility for code extraction from circular definitions based on the combinator Fix. Due to space limitations, several results can only be summarized. The details can be found in the technical appendix.[1]

[1] Appendix available fom `http://arthur.chargueraud.org/research/2010/fix/`

2 Ingredients

2.1 Contraction Conditions for Recursive Functions

The recursion theorem from set theory justifies the existence of a unique function f such that $\forall x.\, f\, x = F\, f\, x$, provided $F\, f\, x$ depends only on the values of $f\, y$ for $y < x$ (with respect to some well-founded relation $<$). The *contraction condition* formally captures the hypothesis of the recursion theorem. In the context of machine-checked proofs, it seem to have first been exploited by Harrison [11].

Definition 1 (Contraction Condition for Recursive Functions). *Let F be a functional of type $(A \to B) \to (A \to B)$, and $<$ be a well-founded relation on values of type A. The* contraction condition *for F with respect to $<$ states:*

$$\forall x\, f_1\, f_2.\, (\forall y < x.\, f_1\, y = f_2\, y) \;\Rightarrow\; F\, f_1\, x = F\, f_2\, x$$

This contraction condition ensures the existence of a unique fixed point for F as soon as the codomain of the recursive function, the type B, is inhabited.

For example, consider the functional as $F\, f\, x \triangleq \mathsf{if}\ x \leq 1\ \mathsf{then}\ 0\ \mathsf{else}\ 1 + f\, \lfloor \frac{x}{2} \rfloor$, which describes a function that computes the binary logarithm of its argument. Let us prove that this functional is contractive. Given arbitrary x, f_1 and f_2 and the assumption "$\forall y < x.\, f_1\, y = f_2\, y$", the proof obligation is:

$$\mathsf{if}\ x \leq 1\ \mathsf{then}\ 0\ \mathsf{else}\ 1 + f_1\, \lfloor \frac{x}{2} \rfloor \quad = \quad \mathsf{if}\ x \leq 1\ \mathsf{then}\ 0\ \mathsf{else}\ 1 + f_2\, \lfloor \frac{x}{2} \rfloor$$

If x is less or equal to 1, then the goal is trivial. Otherwise, we need to show that $f_1\, \lfloor \frac{x}{2} \rfloor$ and $f_2\, \lfloor \frac{x}{2} \rfloor$ are equal. The only way to prove this equality is to use the assumption "$\forall y < x.\, f_1\, y = f_2\, y$". So, we have to justify the fact that $\lfloor \frac{x}{2} \rfloor$ is less than x, which is true because x is greater than one. The inequation $\lfloor \frac{x}{2} \rfloor < x$ indeed captures the fact that the recursive call is made on a value smaller than the current argument x.

Contraction conditions support reasoning on higher-order recursion. They can also be adapted to n-ary recursive functions and mutually-recursive functions, which can be encoded into simple functions using products and sums, respectively. Moreover, contraction conditions can be easily extended so as to support partial functions by restricting arguments to be in a given domain D. For a functional F contractive on a domain D, the fixed point theorem guarantees the existence of a partial fixed point f on that domain, satifying $\forall x.\, D\, x \Rightarrow f\, x = F\, f\, x$. Notice that the use of this theorem requires one to provide the domain D *before* constructing the fixed point f of F.

2.2 Inductive Invariants

As Krstić and Matthews [14] point out, the contraction condition for recursive function fails to handle the case of nested recursion. Consider the nested zero function, described by the functional $F\, f\, x \triangleq \mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ f(f(x-1))$. Trying to prove F contractive leads to the proof obligation $f_1(f_1(x-1)) =$

$f_2(f_2(x-1))$. The hypothesis of the contraction condition can be used to prove $f_1(x-1)$ equal to $f_2(x-1)$, because $x-1$ is smaller than x. However, we have no assumption at all on the value of $f_1(x-1)$, so we cannot prove the equality $f_1(f_1(x-1)) = f_2(f_1(x-1))$. To address this limitation, Krstić and Matthews [14] introduced the notion of *inductive invariants* and used it to weaken the contraction condition, thereby obtaining a stronger fixed point theorem able to handle nested recursion.

Definition 2 (Inductive invariants). *A binary relation S of type $A \to B \to \mathsf{Prop}$ is an* inductive invariant *for a functional F of type $(A \to B) \to (A \to B)$ if there exists a well-founded relation $<$ such that*

$$\forall x\, f.\ (\forall y < x.\ S\, y\, (f\, y)) \Rightarrow S\, x\, (F\, f\, x)$$

The first observation to be made is that if S is an inductive invariant for F, then any fixed point f of F admits S as post-condition, in the sense that $S\, x\, (f\, x)$ holds for any x. Formally, the *restricted contraction condition* for a functional F, with respect to an inductive invariant S, is similar to the contraction condition except that it includes an extra hypothesis about the function f_1. This condition guarantees the existence and uniqueness of a fixed point.

Definition 3 (Restricted Contr. Condition for Recursive Functions)

$$\forall x\, f_1\, f_2.\ (\forall y < x.\ f_1\, y = f_2\, y) \wedge (\forall y.\ S\, y\, (f_1\, y)) \Rightarrow F\, f_1\, x = F\, f_2\, x$$

By instantiating S as the predicate "$\lambda x\, r.\, (r = 0)$", one can prove that the nested zero function admits a unique fixed point and always returns zero.

2.3 Complete Ordered Families of Equivalences

The contraction conditions described so far can only deal with recursion, for the basic reason that recursive calls must be applied to smaller values with respect to a well-founded relation. In order to deal with corecursive functions, Matthews [17] introduced a different form of contraction conditions stated in terms of *families of converging equivalence relations*. Di Gianantonio and Miculan [9] slightly simplified this structure, calling it *complete ordered families of equivalence*, abbreviated as "c.o.f.e.". We follow their presentation.

The contraction condition for a functional F of type $(A \to A) \to A$ is stated in terms of a family of equivalence relations over values of type A, written $\overset{i}{\approx}$, indexed with values of an ordered type I. This family needs to be *complete* in the sense that all *coherent sequences* converge to some limit. Note: the definitions of coherence and of completeness can be skipped upon first reading.

Definition 4 (Ordered Families of Equivalence). *The structure $(A, I, \prec, \overset{i}{\approx})$ is an* ordered family of equivalences *when $\prec$ is a well-founded transitive relation over the type I and $\overset{i}{\approx}$ is an equivalence relation over the type A for any i of type I. Thereafter, the intersection of all the relations $\overset{i}{\approx}$ is written $\approx$.*

Definition 5 (Coherent Sequences). *A sequence u_i of values of type A indexed by elements of type I is said to be* coherent *if for any indices i and j such that $i \prec j$ the values u_i and u_j are equivalent at level i, that is, $u_i \overset{i}{\approx} u_j$. More generally, the sequence u_i is said to be* coherent on the domain K*, for a predicate K of type $I \to \mathsf{Prop}$, when the property $u_i \overset{i}{\approx} u_j$ holds for any i and j satisfying K and such that $i \prec j$ holds.*

Definition 6 (Completeness for an Ordered Family of Equivalences). *An ordered family of equivalences $(A, I, \prec, \overset{i}{\approx})$ is said to be* complete *if, for any downward-closed domain K (i.e., such that $i \prec j$ and $K\,j$ imply $K\,i$) and for any sequence u_i coherent on the domain K, the sequence u_i admits a limit l on the domain K, in the sense that $u_i \overset{i}{\approx} l$ holds for any i satisfying K.*

A basic example of c.o.f.e. is the one associated with streams. In this case, I is the set of natural numbers ordered with $<$. The relation $\overset{i}{\approx}$ relates any two streams that agree up to their i-th element. The intersection $\approx$ of the family of relations $(\overset{i}{\approx})_{i \in \mathbb{N}}$ corresponds to stream bisimilarity. This construction of a c.o.f.e. for sterams can be easily generalized to coinductive trees.

Complete ordered families of equivalences are used to state the following sufficient condition for the existence of a unique fixed point for F modulo $\approx$.

Definition 7 (Contraction Condition for c.o.f.e.'s). *The functional F is* contractive *w.r.t. a complete ordered family of equivalences $(A, I, \prec, \overset{i}{\approx})$ when*

$$\forall x\, y\, i.\ (\forall j \prec i.\ x \overset{j}{\approx} y) \Rightarrow F\,x \overset{i}{\approx} F\,y$$

In the particular case of streams, the contraction condition expresses the fact that if x and y are two streams that agree up to the index $i-1$, then $F\,x$ and $F\,y$ agree up to the index i. More generally, the contraction condition asserts that, given any two values x and y, the functional F is such that $F\,x$ and $F\,y$ are always closer to one another than x and y are, for an appropriate distance.

Di Gianantonio and Miculan [10] have described a general theory, expressed in categories of sheaves, in which complete ordered families of equivalences are simply particular cases of sheaves on well-founded topologies. Their theory also covers the case of well-founded recursion, described by functionals of type $\forall x : A.\, (\{y \mid y < x\} \to B) \to B$. However, di Gianantonio and Miculan do not cover the important case of nested calls, nor do they explain how the contraction condition for recursive functions described by functionals of type $(A \to B) \to (A \to B)$ fits their model.

2.4 Optimal Fixed Point

The combinator Fix_1 for unique fixed points [17] described in the introduction does not work for partial functions because the associated fixed point equation typically admits several partial fixed points. One idea, put forward by Krstić

and Matthews [14] and investigated in more details by Krstić in [13], is that there is always a "best" domain for any functional describing a terminating recursive function, and that, on this domain, there exists a unique fixed point. The formalization of this idea relies on the notion of *inductive fixed point*.

Definition 8 (Inductive Fixed Point). *f is an* inductive fixed point *of a functional F on a domain D if there exists a well-founded relation $<$ such that:*

$$\forall g\, x.\ D\, x \Rightarrow (\forall y < x.\ D\, y \Rightarrow f\, y = g\, y) \Rightarrow f\, x = F\, g\, x$$

Interestingly, an inductive fixed point on a given domain is always the unique fixed point on that domain. Moreover, any functional admits a *maximal* inductive fixed point, which is the inductive fixed point with the largest domain. This theorem, which does not appear to have ever been mechanized, may suggest the definition of a *maximal inductive fixed point combinator*. Such a combinator would be useful for terminating functions. However, it would not accommodate corecursive functions.

In this paper, we invoke an older and much more general theorem in order to formalize the notion of "best" fixed point. The theorem, due to Manna and Shamir [16], asserts the existence of an *optimal fixed point* for any functional describing a partial function. While it was initially designed for recursive programs, the theorem turns out to apply to a much larger class of circular definitions.

Several definitions need to be introduced before we can state this theorem. A *partial function* $\bar{f}$, written with an overline, is represented as a pair (f, D) of a total function f of type $A \to B$ and of a domain D of type $A \to \mathsf{Prop}$. We write $A \hookrightarrow B$ the type of partial functions from A to B. Moreover, we write $\mathrm{dom}(\bar{f})$ the right projection of $\bar{f}$ and write $\underline{f}$ (without an overline) the left projection of $\bar{f}$. Two partial functions $\bar{f}$ and $\bar{f}'$ are said to be *equivalent*, written $\bar{f} \overset{\hookrightarrow}{=} \bar{f}'$, if they have the same domain and are extensionally equal on that domain. Moreover, two partial functions $\bar{f}$ and $\bar{f}'$ are said to be *consistent* if they agree on the intersection of their domains. Finally, a partial function $\bar{f}'$ *extends* a partial function $\bar{f}$, written $\bar{f} \sqsubseteq \bar{f}'$, if the domain of $\bar{f}$ is included in the domain of $\bar{f}'$ and if f and f' are extensionally equal on the domain of $\bar{f}$. Note that the relation $\sqsubseteq$ defines a partial order (modulo $\overset{\hookrightarrow}{=}$) on the set of partial functions. The next two definitions formalize the notion of optimal fixed point.

Definition 9 (Generally-Consistent Fixed Points). *Let $\bar{f}$ be a fixed point modulo $\overset{\hookrightarrow}{=}$ (the equivalence between partial functions) of a functional F of type $(A \hookrightarrow B) \to (A \hookrightarrow B)$. The fixed point $\bar{f}$ is said to be a* generally consistent*, written* $\mathsf{generally\_consistent}\, F\, \bar{f}$*, if any other fixed point $\bar{f}'$ of F modulo $\overset{\hookrightarrow}{=}$ is consistent with $\bar{f}$.*

In other words, a generally-consistent fixed point $\bar{f}$ of a functional F is such that, for any other fixed point $\bar{f}'$ of F, the equation $f'(x) = f(x)$ holds for any x that belongs both to the domain of $\bar{f}$ and that of $\bar{f}'$. The contrapositive of this statement asserts that the domain of a generally-consistent fixed point cannot include any point x whose image is not uniquely determined by the fixed point

equation for F. Thus, as argued by Manna and Shamir [16], generally-consistent fixed points are the only genuine solutions of any circular function definition.

Definition 10 (Optimal Fixed Point). *A partial function $\bar{f}$ of type $A \hookrightarrow B$ is the* optimal fixed point *of a functional F of type $(A \hookrightarrow B) \to (A \hookrightarrow B)$ if it is the greatest generally-consistent fixed point of F, with respect to the partial order $\sqsubseteq$ on the set of partial functions.*

In short, the optimal fixed point $\bar{f}$ of a functional F is the generally-consistent fixed point of F with the largest domain. This means that every other generally-consistent fixed point of F is a restriction of $\bar{f}$ to a smaller domain.

Theorem 1 (Optimal Fixed Point Theorem). *For any functional F of type $(A \hookrightarrow B) \to (A \hookrightarrow B)$, where B is inhabited, F admits an optimal fixed point.*

The optimal fixed point theorem appears to have had relatively little impact as a theory of circular *program* definitions, probably because optimal fixed points are not computable in general. Yet, as a foundation for a theory of circular *logical* definitions, we find the optimal fixed point theorem to be the tool of choice.

2.5 Contributions of This Paper

1. By spotting the interest of optimal fixed points for logical circular definitions and by conducting the first formal development of the optimal fixed point theorem, we obtain a proper treatment of partiality for recursive and corecursive functions in higher-order type theory.
2. Using invariants to generalize existing results on complete ordered families of equivalences, we provide the first general method for justifying the well-definiteness of nested corecursive functions. The use of invariants also supports reasoning on certain forms of corecursive values that could not be formalized with previously-existing contraction conditions.
3. By showing that contraction conditions for recursive functions can be obtained as a particular instance of contraction conditions for complete ordered families of equivalences, even when nested calls are involved, we are able to offer a unified presentation of a number of fixed point theorems based on contraction conditions.

3 The Greatest Fixed Point Combinator

3.1 Definition of the Greatest Fixed Point Combinator

The combinator Fix takes as argument an equivalence relation $\equiv$ and a partial order $\lhd$, both defined on values of an inhabited type A. It then takes a functional F of type $A \to A$ and returns the greatest fixed point of F modulo $\equiv$ with respect to $\lhd$, if it exists. Its definition relies on the predicate "$\mathsf{greatest} \prec P\,x$", which asserts that x satisfies P and that x is greater than any other value satisfying P, with respect to $\prec$.

Definition 11 (The Greatest Fixed Point Combinator)

$$\mathit{Fix}\,(\equiv)\,(\lhd)\,F \quad \triangleq \quad \epsilon\, x.\,[\,\mathit{greatest}\,(\lhd)\,(\mathit{fixed\_point\_modulo}\,(\equiv)\,F)\,x\,]$$

Note that Coq can accomodate Hilbert's epsilon operator. It suffices to add the axiom of indefinite description and that of predicate extensionality. The application of the epsilon operator requires a proof that the type A is inhabited. We encapsulate this proof using an inductive data type $\mathsf{Inhabited}$, of sort $\mathsf{Type} \to \mathsf{Prop}$. (Note that proofs of type $\mathsf{Inhabited}\,A$ need not be manipulated explicitly, thanks to the use of Coq's typeclass facility.) Thus, Fix has type:

$$\forall A.\,(\mathsf{Inhabited}\,A) \to (A \to A \to \mathsf{Prop}) \to (A \to A \to \mathsf{Prop}) \to (A \to A) \to A$$

3.2 Instantiation as a Unique Fixed Point Combinator

The unique fixed point combinator Fix_1, useful for circular definitions that do not involve partial functions, can be defined in terms of Fix. To that end, it suffices to instantiate both $\equiv$ and $\lhd$ as the equality between values of type A.

Definition 12 (Another Unique Fixed Point Combinator)

$$\mathit{FixVal}\,F \quad \triangleq \quad \mathit{Fix}\,(=)\,(=)\,F$$

FixVal is provably equivalent to the definition $\epsilon\, x.(\forall y.\ y = F\,y \iff y = x)$.

More generally, we can construct a combinator for unique fixed point modulo an equivalence relation $\sim$, simply by instantiating both $\equiv$ and $\lhd$ as $\sim$.

Definition 13 (Combinator for Unique Fixed Point Modulo)

$$\mathit{FixValMod}\,(\sim)\,F \quad \triangleq \quad \mathit{Fix}\,(\sim)\,(\sim)\,F$$

3.3 Instantiation as an Optimal Fixed Point Combinator

We now construct a combinator that returns the optimal fixed point of a functional F of type $(A \to B) \to (A \to B)$. First, we need to transform F as a functional between partial functions, of type $(A \hookrightarrow B) \to (A \hookrightarrow B)$, so as to be able to invoke the theory of optimal fixed points. Second, we need to find a suitable instantiation of the relation $\lhd$ to ensure that the greatest fixed point with respect to $\lhd$ is exactly the optimal fixed point. We start with the first task.

Definition 14 ("Partialization" of a Functional). *A functional F of type $(A \to B) \to (A \to B)$ can be viewed as a functional of type $(A \hookrightarrow B) \to (A \hookrightarrow B)$, i.e. as a functional on partial functions, by applying the following "partialization" operator: $\mathsf{partialize}\,F \triangleq \lambda(f, D).\,(F\,f, D)$.*

Definition 15 (Partial Fixed Points). *Given a functional F of type $(A \to B) \to (A \to B)$, we say that $\bar{f}$ is a* partial fixed point *of F if and only if it is a fixed point of the functional "$\mathsf{partialize}\,F$" modulo $\overset{\hookrightarrow}{=}$.*

Our next step is to define a relation $\ll_F$ over the set of fixed points of "partialize F" so that the greatest element of $\ll_F$ is exactly the optimal fixed point of F. On the one hand, the optimal fixed point is a generally-consistent fixed point of "partialize F", moreover it is the greatest with respect to $\sqsubseteq$. On the other hand, the combinator Fix produces a fixed point $\bar{f}$ of "partialize F" which is the greatest with respect to the relation $\ll_F$, meaning that any other fixed point $\bar{f}'$ satisfies $\bar{f}' \ll_F \bar{f}$. To ensure that $\bar{f}$ is the optimal fixed point, we need to ensure (1) that $\bar{f}$ is generally consistent, and (2) that $\bar{f}$ extends any other generally-consistent fixed point. These two requirements give birth to the following definition of $\ll_F$.

Definition 16 (Partial Order Selecting the Optimal Fixed Point)

$$\bar{f}' \ll_F \bar{f} \quad \triangleq \quad \mathit{consistent}\,\bar{f}\,\bar{f}' \wedge (\mathit{generally\_consistent}\,F\,\bar{f}' \Rightarrow \bar{f}' \sqsubseteq \bar{f})$$

Given a functional F of type $(A \to B) \to (A \to B)$, the value returned by "Fix $(\overset{\hookrightarrow}{=})$ $(\ll_F)$ (partialize F)" is a function of type $A \hookrightarrow B$. Since we are not interested in the domain of the resulting function but only in its support, of type $A \to B$, we retain only the first projection.

Definition 17 (The Optimal Fixed Point Combinator)

$$\mathit{FixFun}\,F \quad \triangleq \quad \pi_1(\,\mathit{Fix}\,(\overset{\hookrightarrow}{=})\,(\ll_F)\,(\mathit{partialize}\,F)\,)$$

The following theorem relates the definition of FixFun with that of the optimal fixed point, thereby justifying that FixFun indeed picks an optimal fixed point.

Theorem 2 (Correctness of the Optimal Fixed Point Combinator). *Given a functional F of type $(A \to B) \to (A \to B)$ and a partial function $\bar{f}$ of type $A \hookrightarrow B$, the following two propositions are equivalent:*

1. $\mathit{greatest}\,(\sqsubseteq)\,(\mathit{generally\_consistent}\,F)\,\bar{f}$
2. $\mathit{greatest}\,(\ll_F)\,(\mathit{fixed\_point\_modulo}\,(\overset{\hookrightarrow}{=})\,(\mathit{partialize}\,F))\,\bar{f}$

This ends our construction of the optimal fixed point combinator. The construction can be easily generalized to the case where values from the codomain B are compared with respect to an arbitrary equivalence relation $\equiv$ rather than with respect to Leibniz' equality. This construction results in a strictly more general combinator, called FixFunMod, which is parameterized by the relation $\equiv$.

4 The General Fixed Point Theorem and Its Corollaries

4.1 A General Contraction Theorem for c.o.f.e.'s

Our fixed point theorem for c.o.f.e.'s strengthens the result obtained Matthews [17] and later refined by Di Gianantonio and Miculan [9], adding, in particular, support for nested calls. Our contraction condition generalizes the contraction condition for c.o.f.e.'s with an invariant, in a somewhat similar way as in the restricted contraction condition.

Definition 18 (Contraction Condition). *Given a c.o.f.e.* $(A, I, \prec, \overset{i}{\approx})$*, a functional* F *of type* $A \to A$ *is said to be contractive with respect to an invariant* Q *of type* $I \to A \to \mathsf{Prop}$ *when*

$$\forall x\, y\, i.\ (\forall j \prec i.\ x \overset{j}{\approx} y \wedge Q\, j\, x \wedge Q\, j\, y) \Rightarrow F\, x \overset{i}{\approx} F\, y \wedge Q\, i\, (F\, x)$$

Our fixed point theorem asserts that a given functional admits a unique fixed point as soon as it is contractive with respect to a *continuous* invariant. The notion of continuity that we introduce for this purpose is defined as follows.

Definition 19 (Continuity of an Invariant). *Given a c.o.f.e.* $(A, I, \prec, \overset{i}{\approx})$*, an invariant* Q *is said to be* continuous *if the following implication holds for any downward-closed domain* K*, for any sequence* $(u_i)_{i:I}$ *and for any limit* l*.*

$$(\forall i.\ K\, i \Rightarrow u_i \overset{i}{\approx} l) \wedge (\forall i.\ K\, i \Rightarrow Q\, i\, (u_i)) \Rightarrow (\forall i.\ K\, i \Rightarrow Q\, i\, l)$$

Theorem 3 (Fixed Point Theorem for c.o.f.e.'s). *If* $(A, I, \prec, \overset{i}{\approx})$ *is a c.o.f.e. and if* F *is a functional of type* $A \to A$ *contractive with respect to a continuous invariant* Q *in this c.o.f.e., then* F *admits a unique fixed point* x *modulo* $\approx$*. Moreover, this fixed point* x *is such that the invariant* $Q\, i\, x$ *holds for any* i*.*

The proof of this theorem is fairly involved. The fixed point is constructed as a limit of a sequence, defined by well-founded induction on $\prec$. Each element of this sequence is itself defined in terms of a limit of the previous elements in the sequence. Moreover, the convergence of all those limits depend on the fact that the i-th value of the sequence satisfies the invariant at level i, that is, the predicate $Q\, i$.

4.2 Fixed Point Theorem for Corecursive Values

When F is a contractive functional modulo $\approx$, it admits a unique fixed point modulo $\approx$ (by Theorem 3), thus "$\mathsf{FixValMod}\,(\approx)\,F$" satisfies the fixed point equation for F.

Theorem 4 (Fixed Point Theorem for FixValMod)

$$\begin{cases} x = \mathit{FixValMod}\,(\equiv)\,F \\ (A, I, \prec, \overset{i}{\approx})\ \textit{is a c.o.f.e.} \\ \equiv\ \textit{is equal to}\ \bigcap_{i:I} \overset{i}{\approx} \\ F\ \textit{is contractive w.r.t.}\ Q \\ Q\ \textit{is continuous} \end{cases} \Rightarrow \begin{cases} x \equiv F\, x \\ \forall i.\ Q\, i\, x \end{cases}$$

Compared with previous work, the use of an invariant in the contraction condition makes it strictly more expressive. For the sake of presentation, we consider a simple example. The circular definition associated with the functional $F\, s \triangleq 1 :: (\mathsf{filter}\,(\geq a)\, s)$ is correct only if $a \leq 1$. When this is the case, we can prove F contractive. It suffices to define the invariant Q in such a way that "$Q\, i\, s$" implies that the i first elements of s are greater than or equal to a.

4.3 Fixed Point Theorem for Recursive Functions

The goal of this section is to build a c.o.f.e. that can be used to prove that a functional F of type $(A \to B) \to (A \to B)$ describing a terminating recursive function on a domain D admits a unique fixed point of type $A \to B$. This relatively simple construction, which allows us to unify the various forms of contraction conditions, does not seem to have appeared previously in the literature.

Theorem 5 (c.o.f.e. for Recursive Functions). *Let $\equiv$ be an equivalence relation of type $A \to A \to \mathsf{Prop}$, let $<$ be a well-founded relation of type $A \to A \to \mathsf{Prop}$, and let D be a domain of type $A \to \mathsf{Prop}$. Then, the structure $(A \to B, A, <^+, \overset{x}{\approx})$ is a complete ordered family of equivalences, where $(\overset{x}{\approx})_{x:A}$ is a family of equivalence relations on values of type $A \to B$ defined as follows:*

$$f_1 \overset{x}{\approx} f_2 \quad \triangleq \quad \forall y <^* x.\, D\, y \;\Rightarrow\; f_1\, y \equiv f_2\, y$$

Above, $<^+$ is the transitive closure of $<$ and $<^*$ its reflexive-transitive closure.

In this particular c.o.f.e., the contraction condition can be reformulated in a way which, in practice, is equivalent to the conjunction of the propositions "S is an inductive invariant for F" and "F satisfies the restricted contraction condition with respect to S" (Definition 2 and Definition 3).

Theorem 6 (Contraction Condition for Recursive Functions). *Let D be a domain of type $A \to \mathsf{Prop}$ and let S be a post-condition of type $A \to B \to \mathsf{Prop}$ compatible with $\equiv$, in the sense that if "$S\, x\, y_1$" holds and if "$y_1 \equiv y_2$" then "$S\, x\, y_2$" also holds. Then, in the c.o.f.e. for recursive functions, a functional F is contractive w.r.t. the invariant "$\lambda x\, f.\, D\, x \Rightarrow S\, x\, (f\, x)$" as soon as F satisfies*

$$\begin{array}{l}\forall x\, f_1\, f_2.\, D\, x \wedge (\forall y < x.\, D\, y \;\Rightarrow\; f_1\, y \equiv f_2\, y \wedge S\, y\, (f_1\, y)) \\ \qquad \Rightarrow\; F\, f_1\, x \equiv F\, f_2\, x \wedge S\, x\, (F\, f_1\, x)\end{array}$$

A corollary, not shown here, to the general fixed point theorem (Theorem 3) can be stated for this reformulated contraction condition. The conclusion of this corrolary asserts the existence of a partial fixed point f modulo $\equiv$ on the domain D. Moreover, this fixed point f satisfies the post-condition $\forall x.\, D\, x \Rightarrow S\, x\, (f\, x)$.

The next key theorem in our development establishes that the partial fixed point (f, D) is a *generally-consistent* fixed point of the functional "$\mathsf{partialize}\, F$". The proof of this theorem is quite technical. It reuses and generalizes several ideas coming from the proof that inductive fixed points are generally-consistent [13].

Combining the existence of a generally-consistent fixed point f for F on the domain D with the existence of an optimal fixed point for F, we deduce that the domain of the optimal fixed point of F contains D. It follows that the optimal fixed point for F satisfies the fixed point equation on the domain D.

Theorem 7 (Specification of FixFunMod for Recursive Functions)

$$\left\{\begin{array}{l} f = \mathit{FixFunMod}\,(\equiv)\, F \\ \equiv \textit{ is an equivalence} \\ < \textit{ is well-founded} \\ S \textit{ is compatible with } \equiv \\ F \textit{ is contractive on } D \textit{ w.r.t. } < \textit{ and } S \textit{ modulo } \equiv \end{array}\right. \Rightarrow \left\{\begin{array}{l} \forall x.\, D\, x \;\Rightarrow\; f\, x \equiv F\, f\, x \\ \forall x.\, D\, x \;\Rightarrow\; S\, x\, (f\, x) \end{array}\right.$$

4.4 Fixed Point Theorem for Mixed Recursive-Corecursive Functions

Due to space limitations, we skip the description of the c.o.f.e. for simple corecursive functions and directly focus on the strictly more general c.o.f.e. for functions mixing recursion and corecursion. Compared with the construction proposed by Matthews [17], we have added support for partial functions and for nested calls.

Let $A \to B$ be the type of the function to be constructed and let D be the domain on which we want to prove the function well-defined. The values of the input type A are compared with respect to some well-founded relation, written $<$. The values of the coinductive output type B are compared using an existing c.o.f.e. $(B, I, \prec, \overset{i}{\approx})$. The following result explains how to combine $<$ and $\prec$ in order to construct a c.o.f.e. for the function space $A \to B$. The associated contraction condition and the fixed point theorem follow.

Theorem 8 (c.o.f.e. for Mixed Recursive-Corecursive Functions). *The structure* $(A \to B, I \times A, <', \overset{(i,x)}{\approx'})$ *is a c.o.f.e., where* $<'$ *is the lexicographical order associated with the pair of relations* $(\prec, <^+)$ *and where* $(\overset{(i,x)}{\approx'})_{(i,x):I\times A}$ *is a family of equivalence relations on values of type* $A \to B$ *such that*

$$f_1 \overset{(i,x)}{\approx'} f_2 \quad \triangleq \quad \forall (j,y) \leq' (i,x).\ D\,y \Rightarrow f_1\,y \overset{j}{\approx} f_2\,y$$

Theorem 9 (Contraction Condition for Mixed Functions). *Let* D *be a domain of type* $A \to \mathsf{Prop}$. *Let* S *be an indexed post-condition of type* $I \to A \to B \to \mathsf{Prop}$, *compatible with* $\overset{i}{\approx}$ *in the sense that if "*$S\,i\,x\,y_1$*" holds and if "*$\forall j \prec i.\, y_1 \overset{j}{\approx} y_2$*" holds then "*$S\,i\,x\,y_2$*" holds. Then, in the c.o.f.e. for mixed recursive-corecursive functions, a functional* F *is contractive w.r.t. the invariant "*$\lambda(i,x)\,f.\ D\,x \Rightarrow S\,i\,x\,(f\,x)$*" as soon as* F *satisfies the condition:*

$$\begin{aligned}\forall i\,x\,f_1\,f_2.\ D\,x \wedge (\forall (j,y) <' (i,x).\ D\,y \Rightarrow f_1\,y \overset{j}{\approx} f_2\,y \wedge S\,j\,y\,(f_1\,y) \wedge S\,j\,y\,(f_2\,y))\\ \Rightarrow\ F\,f_1\,x \overset{i}{\approx} F\,f_2\,x \wedge S\,i\,x\,(F\,f_1\,x)\end{aligned}$$

Theorem 10 (Specification of FixFunMod for Mixed Functions)

$$\begin{cases} f = \mathit{FixFunMod}\,(\equiv)\,F \\ < \text{ is a well-founded relation} \\ (B, I, \prec, \overset{i}{\approx}) \text{ is a c.o.f.e.} \\ \equiv \text{ is equal to } \bigcap_{i:I} \overset{i}{\approx} \\ F \text{ is contractive on } D \text{ w.r.t. } S \\ S \text{ is compatible with } \overset{i}{\approx} \end{cases} \Rightarrow \quad \begin{cases} \forall x.\ D\,x \Rightarrow f\,x \equiv F\,f\,x \\ \forall i\,x.\ D\,x \Rightarrow S\,i\,x\,(f\,x) \end{cases}$$

Let us apply this theorem to the filter function. Let f be the function FixFunMod $(\equiv)\,F$, where F is the functional defined in §1.1 and $\equiv$ stands for stream bisimilarity. The domain D characterizes streams that contain infinitely many elements

satisfying the predicate P. Two streams from the domain are compared as follows: $s < s'$ holds if the index of the first element satisfying P in s is less than the index of the first element satisfying P in s'. No invariant is needed here, so we define S such that $S\,i\,s\,s'$ always holds. Let us prove F contractive, as in [17]. Assume the argument s decomposes as $x :: s'$. There are two cases. If x satisfies P, then the goal is $x :: (f_1\,s') \overset{i}{\approx} x :: (f_2\,s')$. This fact is a consequence of the assumption $f_1\,s' \overset{i-1}{\approx} f_2\,s'$, which we can invoke because $(i-1, s')$ is lexicographically smaller than (i, s). If x does not satisfy P, the goal is $f_1\,s' \overset{i}{\approx} f_2\,s'$. This fact also follows from the hypothesis of the contraction condition, because (i, s') is lexicographically smaller than the pair (i, s). Note that this relation holds only because the argument s belongs to the domain D. In conclusion, the equation $f\,s \equiv F\,f\,s$ holds for any stream s in the domain D.

5 Code Extraction

Given a formal development carried out in higher-order logic, one can extract a purely functional program by retaining only the computational content and erasing all the proof-specific elements. The extracted code enjoys a partial correctness property with respect to the original development. Note that termination is usually not guaranteed: even a Caml program extracted from a Coq development can diverge [3]. Our definition of Fix relies on Hilbert's epsilon operator, a non-constructive axiom that does not admit an executable counterpart. Nevertheless, it is still possible to translate the constant Fix into a native "let-rec" construct from the target programming language. (A similar technique in described in Bertot and Komendantsky's work [6].)

Our experiments suggest that this extraction leads to efficient and correct programs, with respect to partial correctness. However, a formal justification of our approach is not attempted in this paper. The theory of code extraction is already far from trivial [15] and there exists, as far as we know, no theory able to account for the correctness of code extraction in the presence of user-defined extraction for particular constants. Thus, in this paper we do not attempt a correctness proof and simply point out that extraction can be set up in practice.

In Haskell, where evaluation is lazy by default, the extraction of the constant Fix is very simple: it suffices to translate "Fix" into "$\lambda F.\ \mathsf{let}\, x = F\,x\,\mathsf{in}\, x$". This value has type "$\forall A.\,(A \to A) \to A$", which is appropriate given the type of Fix. The extraction towards OCaml code is slightly trickier: due to the explicit boxing of lazy values, we need to extract the combinator for corecursive values towards a different constant than that used to extract functions.

6 Other Related Work and Future Work

The most closely related work has already been covered throughout §2. In this section, we briefly mention other approaches to circular definitions. Krauss [12] gives a detailed list of papers dealing with recursive function definitions.

The package TFL developed by Slind [19] supports the definition of total recursive functions for which a well-founded termination relation can be exhibited. Building on Slind's ideas, Krauss [12] developed the *function* package, which supports a very large class of partial recursive functions. It relies on the generation of an inductive definition that captures exactly the domain of the recursive function. Contrary to our work, this approach does not support code generation for partial functions (except tail-recursive ones) and does not support corecursion.

The technique of recursion on an ad-hoc predicate, which consists in defining a function by structural induction on an inductive predicate that describes its domain, was suggested by Dubois and Donzeau-Gouge [8] and developed by Bove and Capretta [7]. Later, Barthe et al. [2] used it in the implementation of a tool for Coq. Besides the fact that it relies heavily on programming with dependent types, one major limitation of this approach is that the treatment of nested recursion requires the logic to support inductive-recursive definitions.

Another possibility for defining terminating recursive functions is to work directly with a general recursion combinator [18], using dependently-typed functionals. Balaa and Bertot [1] proved a fixed point theorem in terms of a contraction condition for functions of type "$\forall x : A.\ (\forall y : A.\ R\,y\,x \Rightarrow B\,y) \Rightarrow B\,x$", where R is some well-founded relation. More recently, Sozeau [20] implemented facilities for manipulating subset types in Coq, including a fixed point combinator for functionals of type $\big(\forall x : A.\ \big(\forall y : \{y : A \mid R\,y\,x\}.\ B\,(\pi_1\,y)\big) \Rightarrow B\,x\big) \Rightarrow \forall x : A.\,(B\,x)$. This approach supports higher-order and nested recursion, but only if the inductive invariant of the function appears explicitly in its type.

Bertot [4] has investigated the formalization of the filter function in constructive type theory. This work was later generalized to support more general forms of mixed recursive-corecursive functions [5]. More recently, Bertot and Komendantsky [6] used Knaster-Tarski's least fixed point theorem as a basis to define and reason on non-terminating functions (the development requires classical logic and a description axiom). The main limitation of their approach is that the output type of every function must be turned into an option type.

In the future, we would like to implement a generator for automatically deriving corollaries to the general fixed point theorem, covering each possible function arity and providing versions with and without domains and invariants. Proving such corollaries by hand on a per-need basis is generally manageable, but having a generator would certainly be much more convenient.

References

1. Balaa, A., Bertot, Y.: Fix-point equations for well-founded recursion in type theory. In: Aagaard, M., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 1–16. Springer, Heidelberg (2000)
2. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 114–129. Springer, Heidelberg (2006)
3. Barthe, G., Frade, M.J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical Structures in Computer Science 14(1), 97–141 (2004)

4. Bertot, Y.: Filters on coinductive streams, an application to eratosthenes' sieve. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 102–115. Springer, Heidelberg (2005)
5. Bertot, Y., Komendantskaya, E.: Inductive and Coinductive Components of Corecursive Functions in Coq. In: Proceedings of CMCS'08, April 2008. ENTCS, vol. 203, pp. 25–47 (2008)
6. Bertot, Y., Komendantsky, V.: Fixed point semantics and partial recursion in Coq. In: Antoy, S., Albert, E. (eds.) PPDP, pp. 89–96. ACM, New York (2008)
7. Bove, A., Capretta, V.: Nested general recursion and partiality in type theory. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 121–135. Springer, Heidelberg (2001)
8. Dubois, C., Donzeau-Gouge, V.: A step towards the mechanization of partial functions: domains as inductive predicates. In: CADE-15 Workshop on mechanization of partial functions (1998)
9. Di Gianantonio, P., Miculan, M.: A unifying approach to recursive and co-recursive definitions. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 148–161. Springer, Heidelberg (2003)
10. Di Gianantonio, P., Miculan, M.: Unifying recursive and co-recursive definitions in sheaf categories. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 136–150. Springer, Heidelberg (2004)
11. Harrison, J.: Inductive definitions: Automation and application. In: Schubert, E.T., Alves-Foss, J., Windley, P. (eds.) TPHOLs 1995. LNCS, vol. 971, pp. 200–213. Springer, Heidelberg (1995)
12. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning (to appear, December 2009)
13. Krstić, S.: Inductive fixpoints in higher order logic (February 2004)
14. Krstić, S., Matthews, J.: Inductive invariants for nested recursion. In: Basin, D.A., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 253–269. Springer, Heidelberg (2003)
15. Letouzey, P.: Programmation fonctionnelle certifiée: L'extraction de programmes dans l'assistant Coq (June 1, 2007)
16. Manna, Z., Shamir, A.: The theoretical aspects of the optimal FixedPoint. SIAM Journal on Computing 5(3), 414–426 (1976)
17. Matthews, J.: Recursive function definition over coinductive types. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 73–90. Springer, Heidelberg (1999)
18. Nordström, B.: Terminating general recursion. BIT 28(3), 605–619 (1988)
19. Slind, K.: Reasoning about Terminating Functional Programs. PhD thesis, Institut für Informatik, Technische Universität München (1999)
20. Sozeau, M.: Subset coercions in COQ. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 237–252. Springer, Heidelberg (2007)

Formal Study of Plane Delaunay Triangulation

Jean-François Dufourd[1] and Yves Bertot[2,⋆]

[1] Université de Strasbourg
LSIIT, UMR CNRS-UdS 7005,
Pôle API, Boulevard S. Brant, BP 10413, 67412 Illkirch, France
dufourd@lsiit.u-strasbg.fr
[2] INRIA-Centre de Sophia Antipolis Méditerranée,
2004, Route des Lucioles, 06902 Sophia-Antipolis Cedex, France
Yves.Bertot@sophia.inria.fr

Abstract. This article presents the formal proof of correctness for a plane Delaunay triangulation algorithm. It consists in repeating a sequence of edge flippings from an initial triangulation until the Delaunay property is achieved. To describe triangulations, we rely on a combinatorial hypermap specification framework we have been developing for years. We embed hypermaps in the plane by attaching coordinates to elements in a consistent way. We then describe what are legal and illegal Delaunay edges and a flipping operation which we show preserves hypermap, triangulation, and embedding invariants. To prove the termination of the algorithm, we use a generic approach expressing that any non-cyclic relation is well-founded when working on a finite set.

1 Introduction

Delaunay triangulation is one of the cornerstones of computational geometry. In two dimensions, the task is, given a collection of input points, to find triangles whose corners are the input points, so that none of the input points lies inside the circumcircle of a triangle. This constraint about circumcircles makes it possible to ensure that flat triangles are avoided as much as possible. This is important for many numeric simulation applications, as flatter triangles imply more errors in the simulation process.

To our knowledge, this article presents the first formalized proof of correctness of an algorithm to build a plane Delaunay triangulation. The algorithm takes as input an arbitrary triangulation and repeatedly flips illegal edges until the Delaunay criterion is achieved. This is one of the most naive algorithms, but proving its formal correctness is already a challenge. We shall review more related work around geometry, combinatorial maps, and formalization in section 2.

We use a general data-structure to represent plane subdivisions and perform proofs, known as hypermaps [32,10,16,3,12,14,15]. Hypermaps are collections of *darts* equipped with two permutations. Darts are elementary objects, more elementary than points: usually, two darts constitute an edge and several darts

⋆ This work is supported by the French ANR project GALAPAGOS (2007-2010).

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 211–226, 2010.

constitutes a point. The two permutations are used to describe how darts are connected together to constitute an edge or a point. We then need to give locations to points. This is done by *embedding* the darts in the plane, by simply attaching coordinates, making sure that all darts that constitute the same point should have the same coordinates. We then restrict our work to *triangulations* by defining a way to compute faces and by considering hypermaps with three-point faces. We shall review our approach to hypermaps in section 3.

The edge flipping operation can be defined at a topological level: it mainly consists in detaching an edge from two points and attaching it back to two other points. As an intermediate step, we observe a hypermap that is not a triangulation, but after re-attaching the edge we get back a new triangulation. We review the topological aspect of edge flipping in section 4.

The next step is to describe where edge flipping occurs. At this point the coordinates of points play a role. We formalize how oriented triangles and circumcircles are computed and define *illegal edges*. We show that illegal edges can be flipped and that the operation also preserves the geometric constraints of well-embedded triangulations. We study this aspect in section 5.

A crucial aspect of our formalization is to show that the algorithm terminates. We tackle this issue by formalizing the argument that the number of possible triangulations based on a given collection of darts and a given collection of points is finite. We then exhibit a real number associated to each triangulation that decreases when an illegal edge is flipped. Because the set of possible triangulations is finite, this is enough to ensure termination. This point is studied in a generic manner in section 6.

In section 7, we show the kind of correctness statement that we have proved about our Delaunay algorithm. The full formalization is developed in Coq [4,9]. It covers many different aspects: hypermaps, geometry, termination problems. Because of the size of this paper, we do not enter into details, but the full formalization is available at [17].

2 Related Work

2.1 Geometric Modeling and Delaunay Triangulations

Like [23], we work with a general model of plane subdivisions, based on hypermaps [10] and combinatorial oriented maps [32]. The triangulations of our development are a kind of combinatorial oriented maps.

Triangulations are widely used in computational geometry to model, reconstruct or visualize surfaces. For instance, the CGAL library offers a lot of advanced functionalities about triangulations [7]. Among them, the Delaunay triangulations [23,25,18,2] are very appreciated in applications because their triangles are regular enough to avoid some numerical artefacts. Pedagogical presentation are given in [18,2].

2.2 Formal Specifications and Proofs in Computational Geometry

We work in the Calculus of Inductive Constructions with Coq [4,9]. Related work on the description of geometric algorithms includes [29] also using Coq and [26] using Isabelle. Concerning graphs, [1] gives a model of triangulations restricted to the study of the five color theorem. Hypermaps are also used intensively in [22] for the proof of the four-colour theorem. A detailed comparison is given in [15]. Hypermaps also play a significant role in the formalization of packings by *tame graphs* in the proof of Kepler's conjecture [28].

Other work with close variants of the hypermaps used in this paper are concerned with the formal study of geometric modelling [31], surface classification [11], image segmentation [13], and a discrete form of the Jordan curve theorem [15].

3 Hypermaps

3.1 Mathematical Aspects

Definition 1. *(*Hypermap*)*
(i) A hypermap *is an algebraic structure* $M = (D, \alpha_0, \alpha_1)$*, where* D *is a finite set, the elements of which are called* darts*, and* α_0*,* α_1 *are permutations on* D*.*

Intuitively, darts can be understood as half-edges, the permutation α_0 usually connects the two darts of each edge, and α_1 connects all the darts that meet on the same vertex of a graph. In general, the α_0 permutation could link together an arbitrary number of elements, but in practice, it is usually involutive. Fig. 1 gives an example of a hypermap with only three darts (darts 7, 10, and 11) that are not 0-linked to another one. Such exotic darts may always occur at intermediate stages during manipulation of maps. For all other darts of Fig. 1, the 0-successor of the 0-successor of each dart is the dart itself.

In Fig. 1, α_0 and α_1 are permutations on $D = \{1, \ldots, 11\}$, then $M = (D, \alpha_0, \alpha_1)$ is a hypermap. It is drawn on the plane by associating to each dart a curved arc (here a simple line segment) oriented from a bullet to a small stroke: 0-linked (resp. 1-linked) darts share the same small stroke (resp. bullet). By convention, in the drawings of hypermaps on surfaces, α_k permutations turn *counterclockwise* around strokes and bullets.

3.2 Formal Encoding

We use Coq's datatype declaration mechanism to define a two element type `dim` of dimensions and an infinite type `dart` of darts, with a special dart singled out for later purposes. This special dart is called `nil`. To describe embeddings we also add a type `point` which is a pair of coordinates (real numbers).

Hypermaps are then described by collecting darts and links in a free map linear data structure of type `fmap`:

```
Inductive fmap : Set :=
 V | I (m:fmap)(d:dart)(p:point) | L (m:fmap)(k:dim)(d1 d2:dart).
```

D	1	2	3	4	5	6	7	8	9	10	11
α0	2	1	4	3	6	5	7	9	8	10	11
α1	6	3	9	5	7	1	8	4	2	11	10

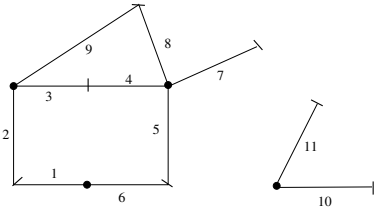


Fig. 1. An example of hypermap embedded on the plane

This defines two operations, `I` to add new dart `d` in an existing map `m`, associating this dart with the location `p`, and `L` to add a link from dart `d1` to dart `d2` in the map `m`, at dimension `k`.

This free data structure is too permissive: we may add the same dart several times, we may link a dart that is not in the map, etc. We will see later that hypermaps are free maps where some preconditions have been verified before adding each dart and link, based on some helper functions.

A first function called `exd` computes whether a given dart is present in a map. Another pair of functions, named `succ` and `pred`, compute whether there is a link at a given dimension with a given dart as source or target. For each dimension, the convention is to include in the free map only links that make up an open path. Thus, to represent a map where $\alpha_k(d_1) = d_2$, $\alpha_k(d_2) = d_3$ and $\alpha_k(d_3) = d_1$, the free map will only contain a link from d_1 to d_2 and a link from d_2 to d_3, or a link from d_2 to d_3 and a link from d_3 to d_1, or a link from d_1 to d_2 and a link from d_3 to d_1. The α_k functions are then computed from the incomplete paths using a recursive algorithm that simply traverses the free map structure. The formal notation in Coq syntax for the α_k functions of a given map m will be `cA m k`.

Hypermaps are then defined as free maps such that some preconditions were verified before applying any of the `I` or `L` operations. The precondition `prec_I` for adding a dart in a hypermap is that the dart should not already be present and should be different from the special dart `nil`. The precondition `prec_L` for adding a link is that the source and the target should be darts in the map, the source should not already have a successor, the target should not already have a predecessor, and the new link should not be closing an open path. As an example of our notations, here is how our `prec_L` function is defined:

```
Definition prec_L (m:fmap)(k:dim)(x y:dart) :=
  exd m x /\ exd m y /\ ~succ m k x /\ ~pred m k y
    /\ cA m k x <> y.
```

Verifying that a free map is indeed a hypermap can be described using a simple recursive function `inv_hmap` that traverses the map and verifies the preconditions at each step:

```
Fixpoint inv_hmap(m:fmap):Prop:=
  match m with
    V => True
```

```
    | I m0 x _ _ => inv_hmap m0 /\ prec_I m0 x
    | L m0 k0 x y => inv_hmap m0 /\ prec_L m0 k0 x y
  end.
```

When `m` is a hypermap, we prove that the α_k, or `cA m k`, are permutations of the darts. Then, by construction, for every dart d the set $\{d' | d' = \alpha_k^n(d)\}$ is finite and is called the *orbit* of d at dimension k. From the most abstract point of view, there is no difference between links at dimension 0 and links at dimension 1. However, to describe the subdivisions we are accustomed to manipulate, it will be better to ensure that orbits at dimension zero are edges, and thus contain only two darts, while orbits at dimension one are vertices, and thus contain only darts that are associated to the same geometrical point (see section 4.2). We also say that two darts x and y are in the same component if there exists a path from x to y using the α_k permutations at each step.

When α_0 and α_1 are permutations, the composition of their inverses $\phi = \alpha_1^{-1} \circ \alpha_0^{-1}$ the orbits of which are the *faces*.

Notions of components, paths, and orbits are independent from the permutation being observed. To handle all these in a regular fashion, we developed a generic module.

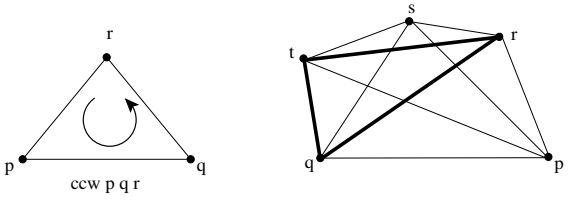
Planar hypermaps can be characterised by counting their edges, vertices, faces, and components [14]. These remain topological properties, independent from actual positions.

Definition 2. *(*Euler characteristic, genus, planarity, Euler formula*)*
Let d, e, v, f, c, be the numbers of darts, edges, vertices, faces, and components of a hypermap.
(i) The Euler characteristic *of M is $\chi = v + e + f - d$.*
(ii) The genus *of M is $g = c - \chi/2$.*
(iii) When $g = 0$, M is said to be planar*. It satisfies the* Euler formula*: $\chi = 2 * c$.*

Truly geometric aspects are described by observing the plane coordinates associated to each dart in the `I` operation. Of course, embeddings are consistent with the geometric intuition only if all darts in a vertex share the same coordinates and the two darts that constitute an edge never have the same coordinates. An extra condition is that faces should not be too twisted: we express this condition only for triangles, by stating that they have to satisfy the *counter-clockwise* orientation predicate as already used by Knuth in [25].

In a nutshell, Knuth's orientation predicate relies on the existence of a 3-argument predicate on points (named `ccw` in our development, Fig. 2(a)) that satisfies five axioms. The first one expresses that if p, q, r satisfy `ccw`, then so do q, r, p (in that order). We shall also use a more complex axiom, which we shall name Knuth's fifth axiom, with the following statement (Fig. 2(b)):

```
Lemma axiom5 :
 forall p q r s t : point,
  ccw q p r -> ccw q p s -> ccw q p t ->
    ccw q r s -> ccw q s t -> ccw q r t.
```



a. Counterclockwise orientation. b. Axiom 5 (result in thick lines).

Fig. 2. Orientation of a triple of points (p, q, r) in the plane and the fifth axiom

Using all these concepts, we can state precisely what we mean by a triangulation: a planar hypermap, where all edges have two darts, and all faces have three vertices. From the geometric point of view, this hypermap should also be well-embedded: all edges contain darts with different geometric locations, all triangles but one are oriented counter-clockwise. The one face that is not a counter-clockwise triangle correspond to the external boundary. In this first experiment, we have assumed this external boundary to also be a triangle, but one that is oriented clockwise (Fig. 3). This simplification can also be found in well-known studies of the Delaunay problem [23]. A hypermap that satisfies all these conditions is said to be a *well-embedded triangulation*.

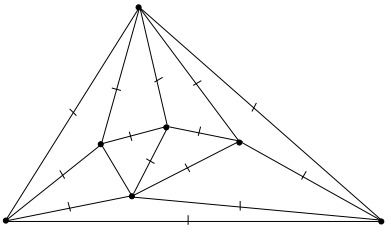


Fig. 3. A triangulation with triangular external face

4 Split, Merge and Flip

In the previous sections, we have described the basic constructors of hypermaps `I` and `L` and the many ways in which we can observe maps and local parts of these maps. Now, we will study ways to transform maps.

4.1 Splitting a K-Orbit, Merging Two K-Orbits

When flipping edges, we need to detach darts from vertices (1-orbits). A more general point of view is to consider that a vertex is actually split into two parts while respecting the connection order. To understand the required transformations, we need to remember that links are left open in the map structure. Before the split, one dart has no 1-successor, after the split two of the darts taken from the split vertex have no 1-successor. The split operation is specified by stating the two darts that have this property, let's assume these two darts are called x and y (Fig. 4).

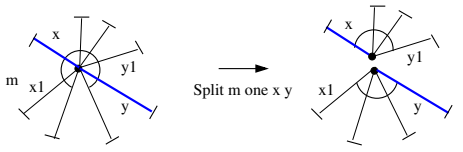


Fig. 4. Splitting a vertex

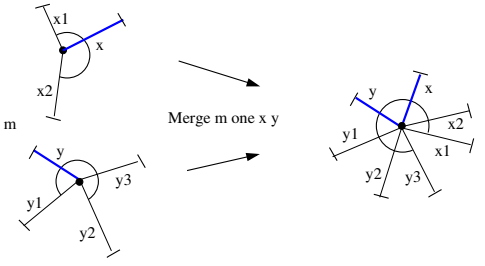


Fig. 5. Merging two vertices

The split operation can be described for any dimension k and is decomposed in two steps. In the first step, one checks whether x has a k-successor. If it has one, then the darts z and t in the k-orbit such that z has no k-successor and t has no k-predecessor are computed, the k-link starting in x is removed, and a link from z to t is added. In this step, the orbit is actually not changed, and we can call this operation *shifting*. In the second step, the one link starting in y is removed. The precondition for this operation is that x and y should be different and in the same k-orbit. In our formal development this is described by a function named `Split` and we proved a few important properties of this operation, for instance that it preserves planarity and commutativity with respect to x and y.

To merge two orbits, we need to choose a dart x in one of the orbit and a dart y in the other, with the intention that the k-successor of x will be y in the new map (Fig. 5). Of course, a first step is to make sure that the two orbits are shifted in such a way that x has no successor and y has no predecessor before adding a link from x to y. This operation has a pre-condition imposing that x and y are not in the same orbit. When considering merging at dimension 1 (merging vertices), the effect on edges and vertices is quite obvious, but less clear for faces [14,16].

4.2 Flipping an Edge

Flipping an edge actually consists in first removing an edge thus "merging" two adjacent triangles, and then adding back a new edge between two different vertices from the merged face. Actually, the two vertices between which a new edge is added are neighbors to the two vertices from which the first edge was

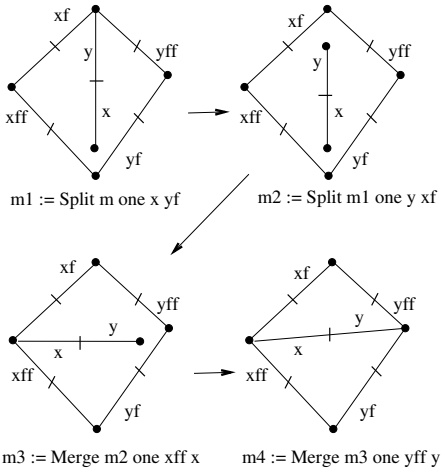


Fig. 6. Four topological steps of Flip

removed. The number of darts in the map is preserved, so that the edge that is removed in the first step can be viewed as moved from a pair of vertices to another one. The first step of removing an edge is described using two split operations, while the second step of adding back a new edge is described using two merge operations. Embeddings must then be updated to respect the requirement that all darts in a vertex share the same location.

The topological steps are illustrated in Fig. 6. The precondition for this operation is that the two darts in the edge should be in different faces and connected to vertices of 3 darts or more.

In intermediate steps, the subdivision is no longer a triangulation: the merged face has a different number of vertices, the detached edge is a component of its own, etc. However, we describe a pair of preconditions, named `prec_Flip` and `prec_Flip_emb` that ensure that the flipping operation as a whole preserves the important topological properties, for instance planarity, having only two-dart edges and three-vertex faces and the embedding properties, for instance that all darts in a 1-orbit (a vertex) share the same coordinates and that all triangles but for the external face are oriented counter-clockwise. The precondition for topological properties (`prec_Flip`) is that the flipped edge consists of darts belonging to distinct faces and to vertices with at least three darts. The precondition for embedding properties (`prec_Flip_emb`) is that the four points in the intermediate merged face should constitute a convex quadrangle. In our formal development, we actually prove that `prec_Flip` is sufficient to preserve the important topological properties, that the `prec_Flip_emb` is sufficient to preserve the well-embedding properties, and that `prec_Flip_emp` implies `prec_Flip` [16]. We shall see that our algorithm for Delaunay triangulation only requires flipping edges that satisfy these predicates.

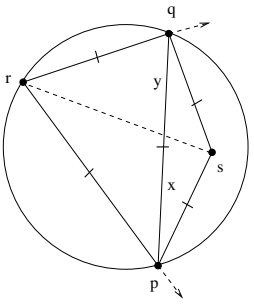


Fig. 7. Point s is in the circumcircle of (p, q, r)

5 The Delaunay Criterion

A triangulation satisfies the Delaunay criterion when none of the vertices occurs inside the circumcircle of a face. In other words, there are no illegal edges. In our development we defined a four-argument predicate `in_circle` to express that a point is inside the circumcircle of three other points.

Definition 3. *(*Illegal edge*)*
An edge is illegal *in a well-embedded plane triangulation when:*
(i) its two adjacent triangles are counterclockwise oriented (which excludes the external face);
(ii) the vertex of one of the two triangles which is not an extremity of the edge is inside the circumcircle of the other triangle.

This notion is illustrated in Fig. 7, where s is inside the circumcircle of triangle (p, q, r), at the right of pq. Note that this property is symmetrical with respect to the two triangles. When an illegal edge is detected, we know that the preconditions for the flip operation are satisfied. When the operation is performed, the new edge produced by this flip operation is legal. This contains two parts: the two new triangles are oriented, and the circumcircles of each new triangle does not contain the fourth point.

More precisely, the important property, called `exchange` in our formal development, asserts that when two adjacent triangles (p, q, r) and (q, p, s) are oriented counterclockwise and s is in the circumcircle of (p, q, r), then the triangles (r, s, q) and (s, r, p) are also oriented counterclockwise (Fig. 7).

Proving this part required some effort. We actually showed that, when p, q, r, and s are in the conditions of the lemma, then there exists a fifth point t so that p, t q, r, and s are in the conditions of Knuth's fifth axiom for the orientation predicate. This point is simply the one obtained by rotating the center of the circumcircle by a quarter-turn around p. We can then use Knuth's fifth axiom to conclude that p, s, r is oriented counterclockwise. A symmetric proof (with a rotation around q) yields that q, r, s is oriented counterclockwise. This symmetric proof is implemented by copying and pasting the formal development, *mutatis mutandi*. Uses of Knuth's first axiom then yield the result.

The 3-argument predicate `ccw` is computed from point coordinates through a simple determinant:

$$\begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix}$$

The boolean condition is represented by the sign of the determinant and the condition of degeneracy, that three points are never aligned, ensures that this determinant is non-zero. The 4-argument predicate, `in_circle` is also computed through the sign of a simple determinant:

$$\begin{vmatrix} x_p & y_p & {x_p}^2 + {y_p}^2 & 1 \\ x_q & y_q & {x_q}^2 + {y_q}^2 & 1 \\ x_r & y_r & {x_r}^2 + {y_r}^2 & 1 \\ x_s & y_s & {x_s}^2 + {y_s}^2 & 1 \end{vmatrix}$$

Knuth's five axioms are easily proved using algebraic tools (in Coq, mostly the `ring` tactic) from these analytic definitions [25,29]. Proving the existence of the point t a few paragraphs above actually relies on a stronger tool, a tactic called `psatz` (the name comes from *positivstellensatz*) and able to handle simple cases of non-linear formulas, available only in recent versions of Coq [6].

6 Termination Based on Finiteness

Traditional approaches to ensure the termination of algorithms rely on structural recursion for the simplest algorithms and well-founded orders for the others. In this work, we took the novel approach of relying on three features:

- We rely on the fact that the number of triangulations embedded on a given finite set of points and using a finite set of darts is finite,
- We exhibit an order on triangulations that is not well-founded, but we show that flipping an illegal edge implies a strict decrease in that order,
- We then rely on the fact that any transitive, irreflexive, and antisymmetric relation R is well-founded when restricted on a finite set.

6.1 A Generic Library for Finiteness

For the formal development, we describe a minimal description of finiteness for subsets of a type. First, we represent each subset of a type `T` by a predicate on `T`, i.e., a function of type `T -> Prop`. Then we express finiteness by stating that all elements satisfying the predicate are found in a list. This is specified by the following datatype declaration:

```
Record fset (T:Type) := mkfs {
  prd :> T -> Prop;
  fs_enum : list T;
  _ : forall x, prd x -> In x fs_enum
}.
```

This declaration states that a finite set on type `T` is described by the characteristic predicate `prd` of type `T -> Prop` and a list `fs_enum` which enumerates the elements that satisfy `prd`. Actually our definition is quite lenient, because it makes it possible to have in the list more elements than those satisfying the predicate. The list is very useful because it gives a simple way to iterate over all the elements in the finite set (and with our lenient definition, risking to see several times the same elements and elements outside the set). This method, of associating two points of view (predicate or covering list) over a simple notion (finite set) is directly inspired from the approach to describe finite sets in the `ssreflect` package [21].

We then show that finiteness is preserved by cartesian product, disjoint sum, inclusion, inverse image through an injection, construction of lists of fixed length, construction of lists of bounded length, and construction of lists without duplication.

To show that the triangulations we consider are in a finite set, we start by computing from any map the list of darts and the set of points that appear in this map. We show that this list of darts and this set of points is preserved during flips. It is easy for the list of darts because the order of the `I` constructors in the `fmap` structure is not modified by the basic shift, split, or merge operations. For points, it is harder, because a flip operation changes the number of darts that use a given coordinate and we need to show that the set is preserved modulo a possible change in the order and number of times each point is inserted. We do this by defining a sorting function with removal of duplicates (an insertion sort algorithm with an extra test to detect duplications) and applying this sorting function on the list of points used in the triangulation. We then show that the list of points obtained after a flip operation, once sorted and cleaned from duplicates, is preserved through flipping.

We then show that all maps built on the same list of darts and the same set of points are in a finite set, obtained using cartesian products, sums, etc.

6.2 A Strict Order on Triangulations

As a complement to the finiteness property, we must exhibit a strict order that decreases every time an illegal edge is flipped.

It is well known that Delaunay triangulation is closely related to computing the three-dimensional convex hull of points projected from the horizontal plane to the revolution paraboloid with equation $z = x^2 + y^2$.

Given four points p, q, r, and s in a three-dimensional space, the determinant obtained from their coordinates by adding a column of ones actually computes a value which is proportional to the volume of the tetrahedron defined by these four points.

$$\begin{vmatrix} x_p & y_p & z_p & 1 \\ x_q & y_q & z_q & 1 \\ x_r & y_r & z_r & 1 \\ x_s & y_s & z_s & 1 \end{vmatrix}$$

Thus, the determinant computed in Section 5 to decide whether a point occurs inside the circumcircle of a triangle actually computes the volume of the tetrahedron defined by the four projections of the points from the plane to the paraboloid. When considering two adjacent triangles and the triangles obtained after flipping the common edge, we can compute the volume between these two triangles and the corresponding triangles using the projected points in the paraboloid. The two configurations yield two different volumes. The difference of volume is exactly the volume of the tetrahedron based on the points in the paraboloid, and it is positive when the projected triangles switch from a concave position to a convex one.

To compute each individual volume, we decompose the prism-like shape into three tetrahedra, each being computed using a determinant. Showing the relation between the volumes of the two prism-like shapes before and after the flip operation and the determinant used for the `in_circle` predicate is an easy task using Coq's ring tool.

To compute the accumulated volume, we simply enumerate the edges of the map and add the triangle obtained as the ϕ-orbit for each edge. Of course, each triangle is thus represented three times, but this does not matter for our decreasing argument. We simply need to show that the volume computed only changes for the six darts whose ϕ-orbit changes during the flip operation.

6.3 Describing a Terminating Function

To describe a terminating function, we rely on a type `tri_map`, which combines a free map and the proof that it is a well-embedded triangulation. This type is defined as a conventional Coq sigma-type:

```
Definition tri_map := {m | inv_Triangulation m /\ isWellembed m}.
```

The natural projection returning the free map is written `p_tri`.

We then define a function `step_tri`, from type `tri_map` to itself, which performs a flip when the map contains an illegal edge. This function relies on the proofs that flip preserves the property of being a well-embedded triangulation. We also define a `final_dec` function that detects when there are no illegal edges.

Last we define a function `nat_measure` which first constructs the final set of all triangulations using the same darts and points, with its enumerating list and then counts the triangulations in this list whose volume is smaller than the current one. This natural number decreases at every flip on a triangulation that contains an illegal edge, i.e., every derivation that does not satisfy the `final` predicate.

The recursive algorithm is not structural recursive, so we need to use one of the tools provided in the Coq system to support general forms of recursion. Here, we use the `Function` command, which accepts a definition as long as one can prove that some measure (a natural number) decreases at each recursive call. We first prove the lemma `non_final_step_decrease` and then provide it to the `Function` command.

```
Lemma non_final_step_decrease :
  forall m, ~final (p_tri m) ->
   (nat_measure (step_tri m) < nat_measure m)%nat.
...

Function delaunay' (t : tri_map) {measure nat_measure} :=
 if final_dec (p_tri t) then
   (p_tri t)
 else
   delaunay' (step_tri t).
```

Computing the finite set of all triangulations is expensive (an exponential cost in the number of darts and points), but this computation is not actually done in the algorithm, it is used as a logical argument for termination. This computation is actually removed from the derived code produced by Coq's extraction facility.

7 Solving the Delaunay Problem

It only makes sense to run the algorithm on well-embedded triangulations. Thus, our `Delaunay` function takes as argument a map and the proofs that this map is a triangulation and that it is well-embedded. It then calls the `delaunay'` function with the adequate element of type `tri_map`.

```
Definition Delaunay (m : fmap)(IT inv_Triangulation m)
    (WE:isWellembed m) : fmap :=
  delaunay' (exist _ m (conj IT WE)).
```

In our formal proof, we show that the end result of the `Delaunay` function returns a well-embedded triangulation that contains no illegal edges. For instance, we have the following statement:

```
Theorem no_dart_illegal_Delaunay :
  forall (m : fmap)(IT: inv_Triangulation m)(WE: isWellembed m),
      no_dart_illegal (Delaunay m IT WE).
```

In English words, we quantify over all free maps that satisfy two predicates. The first predicate `inv_Triangulation` captures all the conditions for the map to be a correct triangulation in the topological sense: it is a correct hypermap, 0-orbits have two elements only, faces have three elements. The second predicate `isWellembedded` expresses that the coordinates are consistent: all darts in the same point share the same coordinates, all triangles are oriented. The hypotheses that the map satisfies these predicates are given names `IT` and `WE` respectively. The function `Delaunay` that computes the Delaunay triangulation takes these hypotheses as arguments. We then use a predicate `no_dart_illegal` to express that the Delaunay condition is always satisfied: it is never the case that the extra vertex of an adjacent triangle is inside the circumcircle of a given triangle.

8 Conclusion

The one missing element of this algorithm is a starter: given an arbitrary set of points inside a triangle, we need to produce the initial triangulation. Developing a naive algorithm, with only the requirement that the triangulation should be well-formed, should be an easy task. Actually, if the three points describing the external face are given first, an possible algorithm is a simple structural recursive function on the list of points.

All numeric computations are described using "abstract perfect" real numbers. In practice, specialists in algorithmic geometry know that numeric computation with floating point numbers can incur failures of the algorithm by failing to detect illegal edges, or by giving inconsistent results for several related computations [33,24]. For instance, rounding errors could make that both an edge and its flipped counterpart could appear to be illegal, thus leading to looping computation that is not predicted by our ideal formal model. However, we know that all predicates are based on determinant computations, hence polynomial computation, and it is thus sufficient to ensure that intermediate computations are done with a precision sufficiently higher than the precision of the initial data to guarantee the absence of errors introduced by rounding. Thus, the "theoretical" correctness of the algorithm can be preserved in a "practical" sense if one relies on a suitable approach to increase the precision of numeric computations, as in [27,30,20].

Our whole development from the hypermap specifications and proofs up to the Delaunay properties reaches about 70,000 Coq lines, with more than 300 definitions and 700 lemmas and theorems. Thanks to the *extraction* facility provided in the Coq sytem, an Ocaml version of the algorithm can be obtained (where every computation on real numbers is replaced by computation on unbound integers for instance, since division is never used in the algorithm) [17].

We described the most naive algorithm for the Delaunay problem. We believe that most of the framework concerning the topology will be re-usable when studying other algorithms for this problem [23,18,2]. Also, our proof reason on abstract models given as Coq programs, not actual programs designed for efficiency. Previous experiments in the formalization of efficient algorithms [5] show that the proofs at an abstract level are a useful first step for the study of efficient programs given in an imperative language.

Our framework is a sound basis for subsequent software developments with triangulations and Flip in computational geometry and geometric modeling, for instance in the way of [3,12,13,8] where hypermaps are represented by linked lists. The functional, side-effect-free approach in this formal description has been very useful for the proofs. However, for efficiency purpose it is crucial to relate this functional description with imperative implementations.

Acknowledgments. We wish to thank L. Pottier, T.-M. Pham, and S. Pion for their suggestions in establishing some of the geometric proofs.

References

1. Bauer, G., Nipkow, T.: The 5 Colour Theorem in Isabelle/Isar. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 67–82. Springer, Heidelberg (2002)
2. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Heidelberg (2008)
3. Bertrand, Y., Dufourd, J.-F.: Algebraic specification of a 3D-modeler based on hypermaps. Graphical Models and Image Processing 56(1), 29–60 (1994)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Text in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
5. Bertot, Y., Magaud, N., Zimmermann, P.: A proof of GMP square root. Journal of Automated Reasoning 29, 225–252 (2002)
6. Besson, F.: Fast Reflexive Arithmetic Tactics: the linear case and beyond. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 48–62. Springer, Heidelberg (2007)
7. Boissonnat, J.-D., Devillers, O., Pion, S., Teillaud, M., Yvinec, M.: Tiangulations in CGAL. Comp. Geom. - Th and Appl. 22(1-3), 5–9 (2002); Spec. iss. SOCG'00 (2002)
8. Brun, C., Dufourd, J.-F., Magaud, N.: Designing and proving correct a convex hull algorithm with hypermaps in Coq. (submitted 2009)
9. The Coq Development Team: The Coq Proof Assistant Reference Manual - Version 8.2. INRIA, France (2009), http://coq.inria.fr
10. Cori, R.: Un Code pour les Graphes Planaires et ses Applications. Astérisque 27 (1970); Société Math. de France
11. Dehlinger, C., Dufourd, J.-F.: Formalizing the trading theorem in Coq. Theoretical Computer Science 323, 399–442 (2004)
12. Dufourd, J.-F., Puitg, F.: Functional specification and prototyping with combinatorial oriented maps. Comp. Geometry - Th. and Appl. 16(2), 129–156 (2000)
13. Dufourd, J.-F.: Design and formal proof of a new optimal image segmentation program with hypermaps. Pattern Recognition 40, 2974–2993 (2007)
14. Dufourd, J.-F.: Polyhedra genus theorem and Euler Formula: A hypermap-formalized intuitionistic proof. Theor. Comp. Sc. 403, 133–159 (2008)
15. Dufourd, J.-F.: An Intuitionistic Proof of a Discrete Form of the Jordan Theorem Formalized in Coq with Hypermaps. Journal of Automated Reasoning 43, 19–51 (2009)
16. Dufourd, J.-F.: Reasoning formally with Split, Merge and Flip in plane triangulations (submitted 2009), http://lsiit.u-strasbg.fr/Publications/index.php
17. Dufourd, J.-F., Bertot, Y.: Formal proof of Delaunay by edge flipping (2010), http://galapagos.gforge.inria.fr/devel/DelaunayFlip.tgz
18. Edelsbrunner, H.: Triangulations and meshes in combinatorial geometry. In: Acta Numerica, pp. 1–81. Cambridge Univ. Press, Cambridge (2000)
19. Flato, E., et al.: The Design and Implementation of Planar Maps in CGAL. WAE 1999 16 (2000); In: Vitter, J.S., Zaroliagis, C.D. (eds.) WAE 1999. LNCS, vol. 1668, pp. 154–168. Springer, Heidelberg (1999)
20. Fousse, L., et al.: MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. 33(2) (2007)
21. 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)

22. Gonthier, G.: Formal proof - the four-Colour theorem. Not. Am. Math. Soc. 55, 1382–1393 (2008)
23. Guibas, L., Stolfi, J.: Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. ACM TOG 4(2), 74–123 (1985)
24. Kettener, L., Mehlhorn, K., Pion, S., Scirra, S., Yap, C.: Classroom examples of robustness problems in geometric computations. Computational Geometry - Theory and Applications 40, 61–78 (2008)
25. Knuth, D.E.: Axioms and Hulls. LNCS, vol. 606. Springer, Heidelberg (1992)
26. Meikle, L.I., Fleuriot, J.: Mechanical Theorem Proving in Computational Geometry. In: Hong, H., Wang, D. (eds.) ADG 2004. LNCS (LNAI), vol. 3763, pp. 1–18. Springer, Heidelberg (2006)
27. Melquiond, G., Pion, S.: Formally Certified Floating-Point Filters For Homogeneous Geometric Predicates. Theoretical Informatics and Applications, EDP Science 41(1), 57–69 (2007)
28. Obua, S., Nipkow, T.: Flyspeck II: the basic linear programs. Annals of Mathematics and Artificial Intelligence 56(3-4), 245–272 (2009)
29. Pichardie, D., Bertot, Y.: Formalizing Convex Hulls Algorithms. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 346–361. Springer, Heidelberg (2001)
30. Priest, D.: Algorithms for Arbitrary Precision Floating Point Arithmetic. In: Tenth Symposium on Computer Arithmetic, pp. 132–143. IEEE, Los Alamitos (1991)
31. Puitg, F., Dufourd, J.-F.: Formal specifications and theorem proving breakthroughs in geometric modelling. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 401–427. Springer, Heidelberg (1998)
32. Tutte, W.E.: Graph Theory. Encyclopedia of Mathematics and its Applications. Addison Wesley, Reading (1984)
33. Yap, C.-K., Pion, S.: Special Issue on Robust Geometric Algorithms and their Implementations. Computational Geometry - Theory and Applications 33(1-2) (2006)

Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison

Amy Felty[1] and Brigitte Pientka[2]

[1] SITE, University of Ottawa, Ottawa, Canada
afelty@site.uottawa.ca
[2] School of Computer Science, McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

Abstract. A variety of logical frameworks support the use of higher-order abstract syntax (HOAS) in representing formal systems given via axioms and inference rules and reasoning about them. In such frameworks, object-level binding is encoded directly using meta-level binding. Although these systems seem superficially the same, they differ in a variety of ways; for example, in how they handle a context of assumptions and in what theorems about a given formal system can be expressed and proven. In this paper, we present several case studies which highlight a variety of different aspects of reasoning using HOAS, with the intention of providing a basis for qualitative comparison of different systems. We then carry out such a comparison among three systems: Twelf, Beluga, and Hybrid. We also develop a general set of criteria for comparing such systems. We hope that others will implement these challenge problems, apply these criteria, and further our understanding of the trade-offs involved in choosing one system over another for this kind of reasoning.

1 Introduction

In recent years, the POPLmark challenge [1] has stimulated considerable interest in mechanizing the meta-theory of programming languages, and the issued problems exercise many aspects that are known to be difficult to formalize. While several solutions have been submitted showing the diversity of possible approaches, it has been hard to compare them. Part of the reason is that while the proposed examples are typical for their domain, they do not highlight the differences between systems. We will bring a different view: As experts in designing and building logical frameworks, we propose a few challenge problems which highlight the differences between different meta-languages, and thereby hopefully provide a better understanding of what practitioners should be looking for.

Our focus in this paper is on encoding meta-theory of programming languages using higher-order abstract syntax (HOAS), where we encode object-level binders with meta-level binders. As a consequence, users can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution, renaming and fresh name generation. Because of this one can think of HOAS encodings as the most advanced technology for specifying programming

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 227–242, 2010.

language meta-theory which leads to very concise and elegant encodings and provides the most support for such an endeavor. However concentrating on encoding binders neglects one important aspect: the support for hypothetical and parametric reasoning. Even in systems supporting HOAS, there is not a clear answer to this. On one side of the spectrum, we find the logical framework Twelf [15] or the dependently-typed functional language Beluga [13,14]. Both systems provide direct support for contexts to keep track of hypotheses. In Twelf, contexts are implicit while in Beluga they are explicit. Supporting contexts directly has two advantages. First, it eliminates the need for building up a context and managing it explicitly via a first-order representation such as a list. More importantly, it eliminates the need to explicitly prove structural properties about contexts, such as weakening. Such built-in support for contexts allows for highly compact proofs. Second, using hypothetical and parametric reasoning provides us with direct meta-level support for applying substitution lemmas. Consequently, substitution lemmas come for free.

On the other side of the spectrum of systems supporting HOAS, we have, for instance, the two-level Hybrid system [10,5] as implemented in Coq [3] and Isabelle/HOL [11], Abella [6], and the Tac prover [2], where contexts are manually represented as lists. While the substitution lemma is still obtained for free because it is an application of the cut-rule, structural properties about contexts such as weakening must typically be proven separately as lemmas. These lemmas can be tedious and they may cause difficulties when automating induction proofs (see [2]). On the other hand, since these systems do not rely on specific built-in procedures for dealing with contexts, there is more flexibility in how they are handled and the necessary reasoning is more transparent to the user. Consequently, proofs in these systems are often easier to understand and to trust.

This paper presents three case-studies which highlight the different treatments of hypothetical reasoning. Along the way, we develop a set of questions which allow a qualitative evaluation and comparison of different reasoning systems. These questions also provide guidance for users and developers in understanding better the differences and limitations. Due to space restrictions, we concentrate on the logical framework Twelf, the functional dependently-typed language Beluga, and the interactive theorem proving environment Hybrid. However, we hope that these problems will subsequently also be implemented using related approaches and serve as a starting point to understand commonalities and differences. Details about the challenge problems and their mechanization can be found in an electronic appendix which is available at `http://complogic.cs.mcgill.ca/beluga/benchmarks`.

2 Examples

In this section, we give an informal presentation and proofs of various properties of the lambda-calculus. We discuss in detail the first example which is concerned with equality reasoning and then briefly sketch the other problems. Formal proofs will be discussed in later sections only for the first. All these examples are purposefully simple, so they can be easily understood and one can

quickly appreciate the capabilities and trade-offs different systems offer. Yet we believe they are representative of the issues and problems arising when formalizing formal systems and proofs about them.

2.1 Equality Reasoning for Lambda-Terms

We begin by defining the syntax of the (untyped) lambda-calculus together with a declarative definition of equality which includes reflexivity and transitivity in addition to the structural rules. We then define the algorithmic version of equality, which concentrates only on the structural rules. We model the declarative definition of equality by the judgment $\Psi \vdash \mathsf{equal}\ M\ N$ and the algorithmic one by the judgment $\Phi \vdash \mathsf{eq}\ M\ N$ and carefully define the contexts Ψ and Φ. The goal is to prove these two versions of equality to be equivalent.

$$\text{Term } M ::= y \mid \mathsf{lam}\, x.\, M \mid \mathsf{app}\ M_1\ M_2 \qquad \begin{array}{l}\text{Context } \Phi ::= \cdot \mid \Phi, \mathsf{equal}\ x\ x \\ \text{Context } \Psi ::= \cdot \mid \Psi, \mathsf{eq}\ x\ x\end{array}$$

Algorithmic Equality

$$\frac{\mathsf{eq}\ x\ x \in \Psi}{\Psi \vdash \mathsf{eq}\ x\ x} \qquad \frac{\Psi, \mathsf{eq}\ x\ x \vdash \mathsf{eq}\ M\ N}{\Psi \vdash \mathsf{eq}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)} \qquad \frac{\Psi \vdash \mathsf{eq}\ M_1\ N_1 \quad \Psi \vdash \mathsf{eq}\ M_2\ N_2}{\Psi \vdash \mathsf{eq}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)}$$

Declarative Equality

$$\frac{\mathsf{equal}\ x\ x \in \Phi}{\Phi \vdash \mathsf{equal}\ x\ x} \qquad \frac{\Phi, \mathsf{equal}\ x\ x \vdash \mathsf{equal}\ M\ N}{\Phi \vdash \mathsf{equal}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)} \qquad \frac{}{\Phi \vdash \mathsf{equal}\ M\ M}$$

$$\frac{\Phi \vdash \mathsf{equal}\ M_1\ N_1 \quad \Phi \vdash \mathsf{equal}\ M_2\ N_2}{\Phi \vdash \mathsf{equal}\ (\mathsf{app}\ M_1\ M_2)\ (\mathsf{app}\ N_1\ N_2)} \qquad \frac{\Phi \vdash \mathsf{equal}\ M\ L \quad \Phi \vdash \mathsf{equal}\ L\ N}{\Phi \vdash \mathsf{equal}\ M\ N}$$

It may be slightly unusual to keep the fact that a variable is equal to itself as a declaration in the context in both formulations. It is only strictly necessary in the first. There are two main reasons. 1) Explicitly introducing the appropriate assumption about each variable is a general methodology which scales to more expressive assumptions. For example, when we specify typing rules, we must introduce a typing context that keeps track of the fact that a given variable has a certain type. 2) Choosing this formulation will also make our proofs more elegant and compact, while at the same time highlight the issues which arise when working with two formal systems each using different assumptions.

We begin by proving that reflexivity and transitivity are indeed admissible from the algorithmic definition of equality.

Theorem 1 (Admissibility of Reflexivity and Transitivity)

1. *If Ψ contains premises for all the free variables in M, then $\Psi \vdash \mathsf{eq}\ M\ M$.*
2. *If $\Psi \vdash \mathsf{eq}\ M\ L$ and $\Psi \vdash \mathsf{eq}\ L\ N$ then $\Psi \vdash \mathsf{eq}\ M\ N$.*

The first theorem can be proven by induction on M. The second can be proven by induction on the first derivation. We now state that when we have a proof for $\mathsf{equal}\ M\ N$ then we also have a proof using algorithmic equality.

Attempt 1 (Completeness). *If* $\Phi \vdash \mathsf{equal}\ M\ N$ *then* $\Psi \vdash \mathsf{eq}\ M\ N$.

However, we note that this statement does not contain enough information about how the two contexts Φ and Ψ are related. In the base case, where we have that $\Phi \vdash \mathsf{equal}\ x\ x$, we must know that for every variable x in Φ there exists a corresponding assumption such that $\mathsf{eq}\ x\ x$ in Ψ. There are two solutions to this problem. 1) We state how two contexts are related and then assume that if this relation holds the theorem holds. 2) We generalize the context used in the theorem such that it contains both assumptions as follows:

$$\text{Generalized context } \Gamma ::= \cdot \mid \Gamma, \mathsf{eq}\ x\ x, \mathsf{equal}\ x\ x$$

where we deliberately state that the assumption $\mathsf{eq}\ x\ x$ always occurs together with the assumption $\mathsf{equal}\ x\ x$, and then apply weakening and strengthening as needed to apply the equality inference rules. Both approaches can be mechanized and we discuss some of the trade-offs later. For now we will concentrate on the latter approach and state the revised generalized theorem.

Theorem 2 (Completeness). *If* $\Gamma \vdash \mathsf{equal}\ M\ N$ *then* $\Gamma \vdash \mathsf{eq}\ M\ N$.

Proof. Proof by induction on the first derivation. We show three cases which highlight the use of weakening and strengthening.

Case 1: Assumption from context
We know $\Gamma \vdash \mathsf{equal}\ x\ x$ where $\mathsf{equal}\ x\ x \in \Gamma$ by assumption. Because of the definition of Γ, we know that whenever we have an assumption $\mathsf{equal}\ x\ x$, we also must have an assumption $\mathsf{eq}\ x\ x$.

Case 2: Reflexivity rule
If the last step applied in the proof was the reflexivity rule $\Gamma \vdash \mathsf{equal}\ M\ M$, then we must show that $\Gamma \vdash \mathsf{eq}\ M\ M$. By the reflexivity lemma, we know that $\Psi \vdash \mathsf{eq}\ M\ M$. By weakening the context Ψ, we obtain the proof for $\Gamma \vdash \mathsf{eq}\ M\ M$.

Case 3: Equality rule for lambda-abstractions

$\Gamma \vdash \mathsf{equal}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)$	by assumption
$\Gamma, \mathsf{equal}\ x\ x \vdash \mathsf{equal}\ M\ N$	by decl. equality rule for lambda-abstraction
$\Gamma, \mathsf{eq}\ x\ x, \mathsf{equal}\ x\ x \vdash \mathsf{equal}\ M\ N$	by weakening
$\Gamma, \mathsf{eq}\ x\ x, \mathsf{equal}\ x\ x \vdash \mathsf{eq}\ M\ N$	by i.h.
$\Gamma, \mathsf{eq}\ x\ x \vdash \mathsf{eq}\ M\ N$	by strengthening
$\Gamma \vdash \mathsf{eq}\ (\mathsf{lam}\, x.\, M)\ (\mathsf{lam}\, x.\, N)$	by alg. equality rule for lambda-abstraction

This proof demonstrates many issues related to the treatment of bound variables and the treatment of contexts. First, we need to be able to apply a lemma which was proven in a context Ψ in a different context Γ. Second, we need to apply weakening and strengthening in the proof. Third, we need to be able to know the structure of the context and we need to be able to take advantage of it. We focus here on these structural properties of contexts, but of course many proofs also need the substitution lemma.

2.2 Reasoning about Variable Occurrences

In this example, we reason about the shape of terms instead of equality of terms. The idea is to compare terms up to variables. For example $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ x\ y$ would have the same shape as $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ y\ x$ but these two terms are obviously not equal. We use the judgment $\Phi \vdash \mathsf{shape}\ M_1\ M_2$ to describe that the term M_1 and the term M_2 have the same shape or structure. Thinking of the lambda-terms being described by a syntax tree, comparing the shape of two terms corresponds to comparing two syntax trees where we do not care about specific variable names which are at the leaves of it. The definition for $\mathsf{shape}\ M_1\ M_2$ can be found in the electronic appendix.

We now prove that if M_1 and M_2 have the same shape, then they must have the same number of variables using the judgment $\Phi \vdash \mathsf{var{-}occ}\ M\ I$ where I describes the total number of variable occurrences in the term M. So for example, the total number of variable occurrences in the term $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ (\mathsf{app}\ y\ x)\ x$ is 3. If we think of the lambda-term as a syntax tree, then I describes the number of leaves in the syntax tree described by the term M. We give three different variations, intended to show differences among systems.

Theorem 3

1. *If* $\Phi \vdash \mathsf{shape}\ M_1\ M_2$
   then there exists an I *such that* $\Phi \vdash \mathsf{var{-}occ}\ M_1\ I$ *and* $\Phi \vdash \mathsf{var{-}occ}\ M_2\ I$. *Furthermore* I *is unique.*
2. *If* $\Phi \vdash \mathsf{shape}\ M_1\ M_2$
   then for all I. $\Phi \vdash \mathsf{var{-}occ}\ M_1\ I$ *implies* $\Phi \vdash \mathsf{var{-}occ}\ M_2\ I$.
3. *If* $\Phi \vdash \mathsf{shape}\ M_1\ M_2$ *and* $\Phi \vdash \mathsf{var{-}occ}\ M_1\ I$ *then* $\Phi \vdash \mathsf{var{-}occ}\ M_2\ I$.

2.3 Reasoning about Subterms in Lambda-Terms

For the next example, we define when a given lambda-term M is a subterm of another lambda-term N and hence we consider M to be structurally smaller than (or equal to) N using the following judgment: $\Psi \vdash \mathsf{le}\ M\ N$. Rules for this judgment are given in the electronic appendix. We concentrate here on stating a very simple intuitive theorem that says that if for all terms N, if N is smaller than K implies that N is also smaller than L, then clearly K is smaller than L.

Theorem 4. *If for all* N. $\Psi \vdash \mathsf{le}\ N\ K$ *implies* $\Psi \vdash \mathsf{le}\ N\ L$ *then* $\Psi \vdash \mathsf{le}\ K\ L$.

This theorem is interesting because in order to state it, we nest quantification and implications placing them outside the fragment of propositions directly expressible in systems such as Twelf.

3 Mechanization in Twelf and Beluga

In this section, we discuss how the previous examples are implemented in Twelf and Beluga. Both systems share an encoding of expressions and inference rules for declarative and algorithmic equality in the logical framework LF [7]. There are several excellent tutorials on how to represent inference rules in the logical framework LF, and hence we keep this very short.

Formalization of Lambda-Terms and Declarative and Algorithmic Equality. Using HOAS, we represent binders in the object-language (see for example $\mathsf{lam}\, x.\, M$) using binders in the meta-language, i.e., the logical framework LF. Hence the constructor `lam` takes in a function of type `exp → exp`. For example, the object-language term $\mathsf{lam}\, x.\, \mathsf{lam}\, y.\, \mathsf{app}\ x\ y$ will be represented in LF as `lam (λx. lam (λy. app x y))`. Bound variables found in the object language, are not explicitly represented in the meta-language.

Object-language	Representation in LF
Term $M ::= y$	`exp : type`
$\mid \mathsf{lam}\, x.\, M$	`lam :(exp → exp) → exp.`
$\mid \mathsf{app}\ M_1\ M_2$	`app : exp → exp → exp.`

We give the implementation of the declarative and algorithmic equality rules next using the two type families `eq` and `equal` respectively. Each inference rule is then represented as a type. Hypothetical derivations (as in the rule for lambda-abstraction) are represented as higher-order functions.

```
eq: exp → exp → type.
eq_lam :  (Πx : exp. eq x x → eq (E x) (F x))
          → eq (lam (λx. E x)) (lam (λx. F x)).
eq_app : eq E1 F1 → eq E2 F2 → eq (app E1 E2) (app F1 F2).

equal: exp → exp → type.
e_l: (Πx:exp. equal x x → equal (T x) (T' x))
     → equal (lam (λx. T x)) (lam (λx. T' x)).
e_a: equal T2 S2  → equal T1 S1  → equal (app T1 T2) (app S1 S2).
e_r: equal T T.
e_t: equal T R → equal R S → equal T S.
```

Proofs as Recursive Functions. Beluga is a functional language where (hypothetical) derivations are characterized by contextual objects and an inductive proof about derivations is written as a recursive function using pattern matching on them. Each case of the proof corresponds to one branch in the function. First, we define the context schema for the context Ψ which was used in defining algorithmic equality to track assumptions of the form $\mathsf{eq}\ x\ x$ (see page 229). Context schemas classify contexts just as types classify terms. It can be defined as follows: `schema eqCtx = block x:exp . eq x x;` This states that our context consists of blocks of assumptions, containing `x:exp` and `eq x x`. More formally, the block-construct introduces a Σ-type grouping the two declarations together.

The reflexivity theorem which stated that for all M there exists a proof for $\mathsf{eq}\ M\ M$ can then be implemented as a recursive function called `ref` which will have the following type: `rec ref : {ψ:(eqCtx)*} {M::exp[ψ]} (eq (M..)(M..))[ψ]`

This can be read as follows: for all contexts ψ which have schema `(eqCtx)*`, for all terms `M`, we have a proof that `(eq (M..) (M..))[ψ]`. Explicit quantification over the context variable ψ is written using curly brackets in `{ψ:(eqCtx)*}`. The schema is annotated with `*` to denote that declarations of the specified schema may be repeated and the context must be passed explicitly by the user. For universally quantifying over `M`, we use curly brackets in `{M::exp[ψ]}`. Central to Beluga is the idea of a contextual type. `M` for example has type `exp[ψ]` which describes an

object `M` which has type `exp` in the context ψ. `M` is hence an expression which may refer to variables in the context ψ. When we use `M` it is associated with a substitution which maps all the variables in ψ to the correct target context. In the example, we use `M` within the contextual type `(eq (M..) (M..))[ψ]`. Hence, `M` is declared in the context ψ and because it is also used in the context ψ, it is associated with the identity substitution, which is written as `..` in our concrete syntax. Intuitively, it means `M` can depend on all the variables which occur in the context described by ψ. The derivation $\Psi \vdash \mathsf{eq}\ M\ M$ is directly captured by the contextual type `(eq (M..) (M..))[ψ]`.

Before we represent the completeness theorem as a recursive function `ceq`, we define the schema of the generalized context, following our previous informal development as follows: `schema eCtx = block x:exp,u:eq x x.equal x x ;`

Finally, we state the type and implementation of the function `ceq`. We indicate that the context γ is implicit in the actual implementation of the proof and will be reconstructed by omitting the `(...)*` when declaring the schema of γ.

```
rec ceq: {γ:eCtx} (equal (T..) (S..))[γ] → (eq (T..) (S..))[γ] =
fn e ⇒ case e of
| [γ] #p.3.. ⇒ [γ] #p.2..                                  % Assumption from context

| [γ] e_r (T.. )⇒ ref [γ] <γ. _ >                          % Reflexivity

| [γ] e_t (D2..) (D1..) ⇒                                  % Transitivity
  let  [γ] F2.. = ceq ([γ] D2..) in
  let  [γ] F1.. = ceq ([γ] D1..) in
    trans ([γ] F1..)  ([γ] F2..)

| [γ] e_l (λx. λu. D.. x u) ⇒                              % Abstraction
  let  [γ,b:block x:exp,u:eq x x . equal x x] F.. b.1 b.2 =
         ceq ([γ, b:block x:exp, u:eq x x . equal x x] D.. b.1 b.3)
  in
     [γ] eq_lam (λx.λv. F.. x v)

| [γ] e_a (D2..) (D1..) ⇒                                   % Application
    let  [γ] F1.. = ceq ([γ] D1..) in
    let  [γ] F2.. = ceq ([γ] D2..) in
      [γ] eq_app (F1..) (F2..) ;
```

We explain the three cases shown also in the proof on page 230. First, let us consider the case where we used an assumption from the context. It is modelled using parameter variables `#p` in Beluga. Operationally, `#p` can be instantiated with any bound variable from the context γ. Since the context γ consists of blocks with the following structure: `block x:exp,u:eq x x . equal x x`, we in fact want to match on the third element of such a block. This is written as `#p.3`... The type of `#p.3` is `equal (#p.1..) (#p.1..)`. Since our context always contains a block and the parameter variable `#p..` describes such a block, we know that there exists a proof for `eq (#p.1..) (#p.1..)` which can be described by `#p.2`...

Second, we consider the case where we applied the reflexivity rule `e_r` as a last step. In this case, we need to refer to the reflexivity lemma we proved about algorithmic equality. To use the function `ref` which implements the reflexivity lemma for algorithmic equality we however need a context of schema `eqCtx` but the context used in the proof for `ceq` is of schema `eCtx`. Since the schema `eCtx` in fact contains at least as much information as the schema `eqCtx`, we should be allowed to pass a context of schema `eCtx` when a context of schema `eqCtx` is

required. This is achieved by incorporating context subsumption in Beluga (see [17,8] for an introduction to context subsumption).

Third, we consider the case for `e_lam`. In this case, we extend the context with the new declarations about variables and pass to the recursive call `ceq` the derivation `[γ, b:block x:exp,u:eq x x.equal x x] D.. b.1 b.3)`. Weakening is built-in. Although the derivation `D` only depends on the context ψ`,x:exp,u:equal x x`, we can use it in the context which also has the assumption `eq x x`. Applying the induction hypothesis corresponds to the recursive call. The result of recursive call is a derivation `F`, where `F` only depends on `x:exp` and `u:eq x x`. In the on-paper proof we employed strengthening. Finally, we use `F` to assemble the final result `eq_lam (λx.λv. F.. x v)`.

The cases where we applied the application rule `e_a` and the transitivity rule `e_t` as a last step are straightforward. In both cases, we simply appeal to the induction hypothesis on the subderivations `D1..` and `D2...` This is implemented as a recursive call to `ceq` using the derivation `[γ] D1..` and the recursive call to `ceq` using the derivation `[γ] D2...` Finally we assemble the result. In the case for applications we use the rule `eq_app` and in the case for transitivity we use the lemma `trans`.

Proofs as Relations. In Twelf, the proof is implemented as a relation between two derivations, and we separately check that it constitutes a total function. The mode declaration says how we must read the relation operationally. The theorem is represented as a type family, and each case of the proof is represented as one type (or clause). The proof is similar to the implementation in Beluga, with a few exceptions. In Twelf, the context in which we prove the theorem is implicit, and there is no generic variable case, but the variable case is folded into the case for lambda-abstraction. We begin by stating the reflexivity theorem as a relation in Twelf together with the corresponding world declaration. Similar to context schemas, world declarations allow us to describe the context in which the theorem is proven. However, unlike schemas, worlds also keep information about base cases. Since variable cases are handled implicitly, not explicitly, the context must not only list assumptions `x:exp` and `u:equal x x` but in addition a proof that reflexivity holds for `x`, i.e., `ref x u`.

```
ref: ΠT:tp.equal T T → type.                %mode ref +T -D.

%block r_block : block {x:term}{u:equal x x}{r_x: ref x u}.
%worlds (r_block) (ref T D).
```

We now inspect the implementation of the proof of the completeness proof from page 230. It will be very similar to our proof in Beluga, except for the treatment of base cases and contexts.

```
ceq: eq T S → equal T S → type.             %mode ceq +E -D.
c_r: ref _ E
     → ceq eq_r E.

c_t: ceq D1 E1 → ceq D2 E2 → tr E1 E2 E
     → ceq (eq_t D2 D1) E.

c_l: (Πx:tm.Πu:equal x x.Πt_x:tr u u u.Πr_x: ref x u.Πv:eq x x.
       ceq v u → ceq (E x v) (D x u))
     → ceq (eq_l E) (eq_l D).
```

```
c_a: ceq F1 D1 → ceq F2 D2
     → ceq (eq_a F2 F1) (equ_a D2 D1).

%block cl:block {x:term}{u:equal x x}{t_x:tr u u u}{r_x:ref x u}{v:eq x x}
           {c_x: ceq v u}.
%worlds (cl) (ceq E D).

%total E (ceq E D).
```

We can read for example the case `c_a` for applications as follows: Given the relation `ceq F1 D1` (i.h. on the derivation `F1` and `D1`) and the relation `ceq F2 D2` (i.h. on the derivation `F2` and `D2`), we know `ceq (e_a F2 F1) (equ_a D2 D1)`. This case is closely related to the case in our functional program. The differences arise in the case for lambda-abstractions. Since Twelf supports contexts only implicitly, we must introduce a variable `x` not only together with the assumption `equal x x` and `eq x x`, as we do in Beluga, but we also must assume that the reflexivity and transitivity lemma hold for this variable and that indeed there is a proof that guarantees that whenever we have `equal x x` we must have a proof for `eq x x`.

Because there is no explicit context and no explicit variable case when reasoning about formal systems, the base cases are scattered and pollute our context. Consequently, it now is harder to compose lemmas and reason about the relationship between different contexts. For example, the world described by blocks `r_block` is not a prefix of the world described by blocks `cl`. In Twelf, this will lead to world subsumption failure and the user needs to weaken manually the proof for reflexivity to include assumptions `t_x:trans u u u`.[1] Apart from the issues around contexts, the Twelf allows a very compact representation of the completeness proof. Weakening and strengthening is handled automatically. For a more detailed explanation regarding the formalization of proofs in the Twelf system and context subsumption, we refer the reader to [8].

4 Mechanization in Two-Level Hybrid

The Hybrid approach [10] exploits the advantages of HOAS within general theorem proving systems. We use a pretty-printed version of Coq concrete syntax in this paper. *Prop* is the type of meta-level formulas and the usual symbols (e.g., $\rightarrow$, $\forall$) represent the meta-level connectives and quantifiers. $[\![A_1; A_2; \ldots; A_n]\!] \rightarrow A$ abbreviates $A_1 \rightarrow (A_2 \rightarrow \cdots (A_n \rightarrow A) \cdots)$, or equivalently $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \rightarrow A$. The symbol $==$ denotes definitional equality. Free variables in inductive definitions and statements of theorems are implicitly universally quantified at the top-level of each clause or statement.

Hybrid provides a type *expr* and a set of operators on this type used to encode object-language syntax. It is built definitionally on the foundation of the meta-language of the underlying theorem prover; no axioms are introduced. The operators that are used in this paper, with their types are:

$$\mathsf{CON} : \mathit{con} \rightarrow \mathit{expr} \mid \mathsf{APP} : \mathit{expr} \rightarrow \mathit{expr} \rightarrow \mathit{expr} \mid \mathsf{LAM} : (\mathit{expr} \rightarrow \mathit{expr}) \rightarrow \mathit{expr}.$$

[1] Alternatively, we can also weaken the transitivity lemma and change the order of blocks.

We define the type *con* later to represent the constants of an object-language.

In the two-level approach used by Hybrid, a specification logic (SL) is defined inductively and used to encode inference rules of object-languages. Hypothetical and parametric judgments are encoded in the SL layer. In this paper, we use a simple SL, a sequent formulation of a fragment of second-order minimal logic with backchaining, adapted from [9] (and also used in [10]). Its syntax and inference rules can be encoded directly as follows:

$$\begin{aligned} \textit{inductive}\ \ oo := {} & \mathsf{tt} : oo \mid \langle \_ \rangle : atm \to oo \mid \_\ \mathsf{and}\ \_ : oo \to oo \to oo \\ & \mid \_\ \mathsf{imp}\ \_ : atm \to oo \to oo \mid \mathsf{all} : (expr \to oo) \to oo \end{aligned}$$

$$\begin{array}{lrcl} \textit{inductive}\ \ \_ \rhd_{\_} \_ : atm\ list \to nat \to oo \to Prop := & & & \\ s\_tt : & & \to & \Gamma \rhd_n \mathsf{tt} \\ s\_and : & [\![\ \Gamma \rhd_n G_1;\ \Gamma \rhd_n G_2\ ]\!] & \to & \Gamma \rhd_{n+1} (G_1\ \mathsf{and}\ G_2) \\ s\_all : & [\![\ (\forall x.\, \mathsf{proper}\, x \to \Gamma \rhd_n G\ x)\ ]\!] & \to & \Gamma \rhd_{n+1} (\mathsf{all}\, x.\, G\ x) \\ s\_imp : & [\![\ A, \Gamma \rhd_n G\ ]\!] & \to & \Gamma \rhd_{n+1} (A\ \mathsf{imp}\ G) \\ s\_init : & [\![\ A \in \Gamma\ ]\!] & \to & \Gamma \rhd_n \langle A \rangle \\ s\_bc : & [\![\ A \longleftarrow G;\ \Gamma \rhd_n G\ ]\!] & \to & \Gamma \rhd_{n+1} \langle A \rangle \end{array}$$

In the inductive definition of *oo*, *atm* is a parameter used to represent atomic predicates of the object-language and $\langle \_ \rangle$ coerces atoms into propositions of type *oo*. In the definition of the SL, we use the symbol $\rhd$ for the sequent arrow and decorate it with natural numbers to allow reasoning by (complete) induction on the *height* of a proof. For convenience we write $\Gamma \rhd G$ if there exists an n such that $\Gamma \rhd_n G$, and furthermore we simply write $\rhd G$ when $\emptyset \rhd G$. The first four clauses of the definition directly encode the introduction rules of a sequent calculus for this logic. Terms of type *expr* are built on an underlying de Bruijn syntax. The use of the proper annotation rules out terms that have occurrences of bound variables that do not have a corresponding binder (*dangling indices*).[2] In the last two rules, atoms are provable either by assumption or via *backchaining* over a set of Prolog-like rules, which encode the properties of the object-language. The notation $A \longleftarrow G$ represents an instance of one of the clauses of this definition. The sequent calculus is parametric in those clauses.

A small set of structural rules of the SL is proved, and used to reason about object-languages. We prefix theorems formalized in Hybrid with "H-."

H-Theorem 5 (Structural Properties)

(*a*) *Height weakening:* $[\![\ \Gamma \rhd_n G;\ n < m\ ]\!] \to \Gamma \rhd_m G$
(*b*) *Context weakening:* $[\![\ \Gamma \rhd_n G;\ \Gamma \subseteq \Gamma'\ ]\!] \to \Gamma' \rhd_n G$
(*c*) *Atomic cut:* $[\![\ A, \Gamma \rhd G;\ \Gamma \rhd \langle A \rangle\ ]\!] \to \Gamma \rhd G$

Formalization of Lambda-Terms and Declarative and Algorithmic Equality. To represent the object-language, we fill in the definition of *con*, define new operators app and lam using the operators defined earlier for *expr*, and fill in the definition of *atm*, which includes the is_tm relation for well-formedness of terms as well as eq and equal. The inference rules are inductively defined using $(\_ \longleftarrow \_)$.

[2] Hybrid 0.2 described in [10] includes an improvement that doesn't require the proper predicate, but the proofs in this paper are not yet ported to the new version.

$$
\begin{array}{l}
\mathit{inductive}\ \mathit{con} := \mathit{cAPP} : \mathit{con} \mid \mathit{cLAM} : \mathit{con} \\
\quad \mathsf{app}\ M_1\ M_2 == (\mathsf{APP}\ (\mathsf{APP}\ (\mathsf{CON}\ \mathit{cAPP})\ M_1)\ M_2) \\
\quad \mathsf{lam}\, x.\, M\ x == (\mathsf{APP}\ (\mathsf{CON}\ \mathit{cLAM})\ (\mathsf{LAM}\ (\lambda x.\, M\ x)) \\
\\
\mathit{inductive}\ \mathit{atm} := \mathsf{is\_tm} : \mathit{expr} \to \mathit{atm} \mid \mathsf{eq}, \mathsf{equal} : \mathit{expr} \to \mathit{expr} \to \mathit{atm} \\
\mathit{inductive}\ \_ \longleftarrow \_ : \mathit{atm} \to \mathit{oo} \to \mathit{Prop} := \\
\mathit{tm\_lam} : \quad [\![\ \mathsf{abstr}\, T\ ]\!] \to \mathsf{is\_tm}\ (\mathsf{lam}\, x.\, Tx) \longleftarrow \mathsf{all}\, x.\, (\mathsf{is\_tm}\ x)\ \mathsf{imp}\ \langle \mathsf{is\_tm}\ (Tx) \rangle \\
\mathit{tm\_app} : \quad \to \mathsf{is\_tm}\ (\mathsf{app}\ T_1\ T_2) \longleftarrow \langle \mathsf{is\_tm}\ T_1 \rangle\ \mathsf{and}\ \langle \mathsf{is\_tm}\ T_2 \rangle \\
\mathit{eq\_lam} : \quad [\![\ \mathsf{abstr}\, E;\ \mathsf{abstr}\, F\ ]\!] \to \mathsf{eq}\ (\mathsf{lam}\, x.\, Ex)\ (\mathsf{lam}\, x.\, Fx) \longleftarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{all}\, x.\, (\mathsf{eq}\ x\ x)\ \mathsf{imp}\ \langle \mathsf{eq}\ (Ex)\ (Fx) \rangle \\
\mathit{eq\_app} : \quad \to \mathsf{eq}\ (\mathsf{app}\ E_1\ E_2)\ (\mathsf{app}\ F_1\ F_2) \longleftarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle \mathsf{eq}\ E_1\ F_1 \rangle\ \mathsf{and}\ \langle \mathsf{eq}\ E_2\ F_2 \rangle \\
\mathit{e\_l} : \quad [\![\ \mathsf{abstr}\, T;\ \mathsf{abstr}\, T'\ ]\!] \to \mathsf{equal}\ (\mathsf{lam}\, x.\, Tx)\ (\mathsf{lam}\, x.\, T'x) \longleftarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{all}\, x.\, (\mathsf{is\_tm}\ x)\ \mathsf{imp}\ (\mathsf{equal}\ x\ x)\ \mathsf{imp}\ \langle \mathsf{equal}\ (Tx)\ (T'x) \rangle \\
\mathit{e\_a} : \quad \to \mathsf{equal}\ (\mathsf{app}\ T_1\ T_2)\ (\mathsf{app}\ S_1\ S_2) \longleftarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \langle \mathsf{equal}\ T_1\ S_1 \rangle\ \mathsf{and}\ \langle \mathsf{equal}\ T_2\ S_2 \rangle \\
\mathit{e\_r} : \quad \to \mathsf{equal}\ T\ T \longleftarrow \langle \mathsf{is\_tm}\ T \rangle \\
\mathit{e\_t} : \quad \to \mathsf{equal}\ T\ S \longleftarrow \langle \mathsf{equal}\ T\ R \rangle\ \mathsf{and}\ \langle \mathsf{equal}\ R\ S \rangle
\end{array}
$$

The well-formedness clauses *tm_lam* and *tm_app* are required since Hybrid terms are untyped (all object-level terms have type *expr*). Each of the remaining clauses of the inductive definition is given the same name as the corresponding rule in the Twelf and Beluga encoding. Note that they are quite similar; the differences in the encodings include 1) the abstr conditions used to rule out meta-level functions that do not encode object-level syntax, and (2) the appearance of is_tm in the *e_l* and *e_r* clauses, which are required to prove *adequacy* of the encoding (see [5] for a fuller discussion of adequacy of Hybrid encodings). In particular, we prove:

$$
\begin{array}{c}
\rhd\ \langle \mathsf{eq}\ T\ S \rangle \to \rhd\ \langle \mathsf{is\_tm}\ T \rangle \wedge \rhd\ \langle \mathsf{is\_tm}\ S \rangle \\
\rhd\ \langle \mathsf{equal}\ T\ S \rangle \to \rhd\ \langle \mathsf{is\_tm}\ T \rangle \wedge \rhd\ \langle \mathsf{is\_tm}\ S \rangle.
\end{array}
$$

Formalization of Completeness for Algorithmic Equality. In place of classifying contexts using context schemas or worlds declarations, we adopt the notion of a *context invariant*. This notion is informal; since we have an expressive logic at our disposal, we can define any predicate on contexts. We present one approach and briefly discuss a second one. In the first, we have three context invariants, one each for the proofs of reflexivity, transitivity, and completeness.

$$
\begin{array}{rl}
\mathsf{ref\_inv}\ \Phi\ \Psi & == (\forall x.\ \mathsf{is\_tm}\ x \in \Phi \to \mathsf{eq}\ x\ x \in \Psi) \\
\mathsf{tr\_inv}\ \Psi & == (\forall x\ y.\ \mathsf{eq}\ x\ y \in \Psi \to x = y) \\
\mathsf{ceq\_inv}\ \Phi\ \Psi & == \mathsf{ref\_inv}\ \Phi\ \Psi \wedge \mathsf{tr\_inv}\ \Psi \wedge (\forall x\ y.\ \mathsf{equal}\ x\ y \in \Phi \to \mathsf{eq}\ x\ y \in \Psi)
\end{array}
$$

Context invariants are used for two purposes here: 1) to represent how two contexts in different judgments are related (e.g., ref_inv), and 2) to represent information contained in the Σ-type groupings found in the `block` declarations in Beluga and Twelf (e.g., tr_inv). If we include enough information, as we do here, no weakening or strengthening is needed in the completeness proof. Instead, the following property is needed.

H-Lemma 6 (Context Extension)
$\mathsf{ceq\_inv}\ \Phi\ \Psi \to \mathsf{ceq\_inv}\ (\mathsf{equal}\ x\ x, \mathsf{is\_tm}\ x, \Phi)\ (\mathsf{eq}\ x\ x, \Psi)$

We now state the reflexivity and completeness theorems, and discuss the proof of the completeness theorem.

H-Theorem 7 (Reflexivity). $[\![\ \mathsf{ref\_inv}\ \Phi\ \Psi;\ \Phi \rhd_n \langle \mathsf{is\_tm}\ T\rangle\ ]\!] \to \Psi \rhd_n \langle \mathsf{eq}\ T\ T\rangle$

In addition to being necessary for adequacy, well-formedness definitions provide a convenient form of induction, which is used to prove the above theorem.

H-Theorem 8 (Completeness)
$[\![\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \Phi \rhd_n \langle \mathsf{equal}\ T\ S\rangle\ ]\!] \to \Psi \rhd_n \langle \mathsf{eq}\ T\ S\rangle$

The proof is by induction on n with induction hypothesis:

$$IH == [\![\ i < n;\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \Phi \rhd_i \langle \mathsf{equal}\ T\ S\rangle\ ]\!] \to \Psi \rhd_i \langle \mathsf{eq}\ T\ S\rangle.$$

A derivation of $\Phi \rhd_n \langle \mathsf{equal}\ T\ S\rangle$ must end in an application of the last two clauses of the definition of the SL (*s_init* or *s_bc*, page 236). In the *s_init* case (the assumption from context case), we know that $(\mathsf{equal}\ T\ S) \in \Phi$. By the definition of ceq_inv, we know that $(\mathsf{eq}\ T\ S) \in \Psi$. We use this fact and simply apply *s_init* to obtain $\Psi \rhd_n \langle \mathsf{eq}\ T\ S\rangle$, as desired.

When the derivation ends in *s_bc*, it must be the case that one of the four clauses defining declarative equality (page 237) was used. We consider reflexivity (*e_r*) and abstraction (*e_l*). In the former, we know that $T = S$ and $\Phi \rhd_{n-1} \langle \mathsf{is\_tm}\ T\rangle$. By H-Theorem 7, we can conclude $\Psi \rhd_{n-1} \langle \mathsf{eq}\ T\ T\rangle$ and by H-Theorem 5(a), that $\Psi \rhd_n \langle \mathsf{eq}\ T\ T\rangle$.

In the abstraction case (*e_l*), we know that T and S have the form $(\mathsf{lam}\, x.\, T'x)$ and $(\mathsf{lam}\, x.\, S'x)$, respectively, and we must show:

$$[\![\ IH;\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \Phi \rhd_n \langle \mathsf{equal}\ (\mathsf{lam}\, x.\, T'x)\ (\mathsf{lam}\, x.\, S'x)\rangle\ ]\!] \to \Psi \rhd_n \langle \mathsf{eq}\ (\mathsf{lam}\, x.\, T'x)\ (\mathsf{lam}\, x.\, S'x)\rangle$$

By repeated inversion of the SL rules on the last premise, and repeated backward application of these rules to the conclusion, we obtain:

$$[\![\ IH;\ \mathsf{ceq\_inv}\ \Phi\ \Psi;\ \mathsf{proper}\, x;\ (\mathsf{equal}\ x\ x, \mathsf{is\_tm}\ x, \Phi) \rhd_{n-4} \langle \mathsf{equal}\ (T'x)\ (S'x)\rangle\ ]\!] \to (\mathsf{eq}\ x\ x, \Psi) \rhd_{n-3} \langle \mathsf{eq}\ (T'x)\ (S'x)\rangle$$

We can conclude $\mathsf{ceq\_inv}\ (\mathsf{equal}\ x\ x, \mathsf{is\_tm}\ x, \Phi)\ (\mathsf{eq}\ x\ x, \Psi)$ by H-Lemma 6 applied to the second premise, and then apply the induction hypothesis to obtain:

$$[\![\ IH;\ \ldots;\ (\mathsf{eq}\ x\ x, \Psi) \rhd_{n-4} \langle \mathsf{eq}\ (T'x)\ (S'x)\rangle\ ]\!] \to (\mathsf{eq}\ x\ x, \Psi) \rhd_{n-3} \langle \mathsf{eq}\ (T'x)\ (S'x)\rangle$$

which is provable directly by an application of H-Theorem 5(a).

We can also prove this theorem using a generalized context as is done in Twelf and Beluga. Using this alternate approach, we have only one context, so the context invariant no longer needs to express relationships between different

contexts; now it only needs to contain the following information, which is also found in the `block` declarations in Beluga and Twelf.

$$\mathsf{ceq\_inv}' \; \Gamma == (\forall xy.\, \mathsf{eq}\ x\ y \in \Gamma \to x = y) \wedge (\forall xy.\, \mathsf{equal}\ x\ y \in \Gamma \to x = y).$$

Using this approach, we must also explicitly define weakening and strengthening functions on contexts, and lemmas about them. Such functions and lemmas depend on the object-language, but the lemmas are fairly easily proved using H-Theorem 5(b), which is the general weakening theorem of the SL. The reasoning required to prove this new version of H-Theorem 8 is similar to before, though slightly complicated by the need to explicitly apply the weakening and strengthening lemmas.

5 Criteria for Comparison

In this section we compare the approach taken in the three systems considered in this paper. More generally, we describe a list of questions which can be used to quantitatively compare systems and highlight their differences.

How do we represent contexts in proofs? Beluga supports explicit contexts when implementing proofs about LF objects. Context variables allow us to abstract over concrete contexts and the structure of contexts is defined by context schemas. Σ-types tie different declarations together. While Beluga shares the general ideas regarding representing and reasoning about contexts with the Twelf system, it makes the meta-theoretic reasoning about contexts which is hidden in Twelf explicit. In Twelf, the actual context of hypotheses remains implicit. In Hybrid, contexts are explicitly modelled using lists or sets in the SL, but do not appear in the specification of the inference rules of the object-language in the inductive definition of $(\_ \longleftarrow \_)$.

How do we reason with contexts? Reasoning with contexts is particularly important when reasoning about the relationship between two formal systems (such as the equality example) and when we assemble larger proofs using lemmas. Systems like Twelf and Beluga also support structural reasoning about contexts; for example, weakening is supported by the underlying typing rules and context subsumption. This built-in support is sensitive to the ordering of elements in a context (or world) schema and may require explicit weakening as in the implementation of the completeness proof for equality in Twelf. In Hybrid, H-Theorem 5 supports simple reasoning about contexts. This theorem is carried out once and for all at the SL level, and reused for every object-language. More complicated reasoning about weakening and strengthening can be avoided in Hybrid by expressing relationships between contexts in different judgments. The trade-off is that we must define these relationships explicitly as context invariants and prove context extension lemmas. The meta-logic, however, provides considerable flexibility in expressing such invariants. In fact, as discussed earlier, we can express them so that they use generalized contexts and lead to proofs that follow the corresponding Twelf and Beluga proofs quite closely. Doing so requires explicit weakening and strengthening lemmas that are specific to

the object-language. Much of the reasoning that uses these kinds of lemmas is stereotyped and could be automated (although we have not yet done so).

How do we retrieve elements from a context? As the context is implicit in Twelf and the user has no access to it, the user cannot pattern match on elements from it directly. Instead of generic cases which pattern match on an element from the context, base cases in proofs are handled whenever an assumption is introduced, and in fact are treated as part of the context. This may lead to scattering of base cases and redundancy, and in addition complicates reasoning about contexts. In Beluga, retrieval is supported using parameter variables and projections. This is in fact crucial in the reflexivity `ref` and in the completeness proof `ceq`, where we use Σ-types to tie assumptions together and use projections on a parameter variable to retrieve different parts. Since the context is explicit in the SL level in Hybrid, when retrieval of elements is needed in the base case and other cases, it is done via simple list operations such as membership. The Coq libraries provide support for using such operations in proofs.

How easy is it to state a given theorem? The examples in sections 2.2 and 2.3 provide a wide range of statements. All discussed systems provide a two-level approach. However, the level which allows reasoning about formal systems is more expressive in Beluga and in Hybrid. These systems provide direct representations of the given theorems. Twelf's meta-logic which is used to verify that a given relation constitutes a proof is not rich enough to handle nested quantification and implications directly. One solution is to implement an assertion logic which is then used to encode the actual theorem [18].

How do we apply a substitution lemma? In all known systems supporting HOAS encodings substitution lemmas come for "free." While the examples in this paper do not make use of the substitution lemma, there are several well-known examples such as type preservation for MiniML. In the Twelf system and in Beluga, applying the substitution lemma is reduced to the substitution operation in the underlying logical framework. In Hybrid, the substitution lemma corresponds to the application of the SL cut-rule, expressed as H-Theorem 5(c).

How do we know we have implemented a proof? In a system such as Hybrid, we simply need to trust the underlying proof assistant and establish adequacy. In general, proofs proceed by induction on the definition of the SL with a subinduction on the object-language. Coq provides extensive support for inductive reasoning, and the induction hypothesis is a premise that is applied explicitly when needed.

In systems such as Twelf or Beluga, we need to establish separately that the user implemented a total function. Twelf is a mature system and provides a coverage checker which in turn relies on the world declarations to ensure the base cases are covered. In addition, the termination checker verifies that a given relation is terminating according to a structural ordering specified by the user. This establishes that all appeals to the induction hypothesis are valid.

Beluga essentially adopts the same philosophy, although the current release does not include a coverage and termination checker. The theoretical foundation

for coverage in Beluga is described in [4] and an implementation is planned for the future. Intuitively, pattern matching on a contextual object of type $A[\Psi]$ is exhaustive if we cover all constructors of type A plus the cases described by parameter variables, which cover the possibility that we have used an assumption from the context Ψ. For termination checking, we believe the ideas from [12] can be easily adapted.

In general, writing cases using pattern matching by hand may result in a more compact proof since it provides a flexible way to write fall-through patterns or to simultaneously match on several objects. Hence, we may get away with writing fewer cases explicitly as compared to an interactive prover.

How easy is it to interface the system with, for example, support for natural numbers? In the example that counts variable occurrences, reasoning about natural numbers may be necessary and useful. Twelf and Beluga's reasoning infrastructure does not support them and hence properties like addition and the totality of addition must be proven separately. This leads to some overhead in the actual proofs. Hybrid, on the other hand, relies heavily on the theorem prover's built in data-type for natural numbers along with a large collection of lemmas and automated proof procedures (such as *omega* in Coq).

6 Conclusion

We presented several benchmark problems together with a general set of criteria for comparing reasoning systems which support the mechanization of formal systems. In addition, we discussed in detail the proofs of one of these problems in three systems (Beluga, Twelf, and Hybrid), and applied these criteria to compare them. This work is a starting point that will help users and developers to evaluate proof assistants which mechanize the reasoning about formal systems. It will also facilitate a better understanding of the differences between and limitations of these systems, as well as the impact of these design decisions in practice. This will provide guidance for users and stimulate discussion among developers.

We hope that these problems will subsequently also be implemented in systems using related approaches. In particular, the Delphin System [16] seems to lie between the three systems discussed in this paper. Similarly, it would be interesting to compare systems such as Abella as well as approaches not relying on HOAS encodings such as nominal encodings.

References

1. Aydemir, B., et al.: Mechanized metatheory for the masses: The POPLmark challenge. In: Hurd, J., Melham, T.F. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
2. Baelde, D., Snow, Z., Miller, D.: Focused inductive theorem proving. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 278–292. Springer, Heidelberg (2010)

3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
4. Dunfield, J., Pientka, B.: Case analysis of higher-order data. In: LFMTP'08. Electr. Notes in Theor. Comput. Sci, vol. 228, pp. 69–84 (2009)
5. Felty, A.P., Momigliano, A.: Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. CoRR, abs/0811.4367 (2008)
6. 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)
7. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)
8. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of Functional Programming 17(4-5), 613–673 (2007)
9. McDowell, R.C., Miller, D.A.: Reasoning with higher-order abstract syntax in a logical framework. ACM Transactions on Computational Logic 3(1), 80–136 (2002)
10. Momigliano, A., Martin, A.J., Felty, A.P.: Two-level Hybrid: A system for reasoning using higher-order abstract syntax. In: LFMTP'07. Electr. Notes Theor. Comput. Sci, vol. 196, pp. 85–93 (2008)
11. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Pientka, B.: Verifying termination and reduction properties about higher-order logic programs. Journal of Automated Reasoning 34(2), 179–207 (2005)
13. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08), pp. 371–382. ACM Press, New York (2008)
14. Pientka, B., Dunfield, J.: 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)
15. 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)
16. Poswolsky, A.B., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 93–107. Springer, Heidelberg (2008)
17. Schürmann, C.: Automating the Meta Theory of Deductive Systems. PhD thesis, Department of Computer Science, Carnegie Mellon University. CMU-CS-00-146 (2000)
18. Schürmann, C., Sarnat, J.: Structural logical relations. In: 23rd Annual Symposium on Logic in Computer Science (LICS), pp. 69–80. IEEE Computer Society, Los Alamitos (2008)

A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture

Anthony Fox and Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. This paper presents a new HOL4 formalization of the current ARM instruction set architecture, ARMv7. This is a modern RISC architecture with many advanced features. The formalization is detailed and extensive. Considerable tool support has been developed, with the goal of making the model accessible and easy to work with. The model and supporting tools are publicly available – we wish to encourage others to make use of this resource. This paper explains our monadic specification approach and gives some details of the endeavours that have been made to ensure that the sizeable model is valid and trustworthy. A novel and efficient testing approach has been developed, based on automated forward proof and communication with ARM development boards.

1 Introduction

Instruction set architectures (ISAs) provide a precise interface between hardware (microprocessors) and software (high-level code). Formal models of instruction sets are pivotal when verifying computer micro-architectures and compilers. There are also areas where it is appropriate to reason directly about low-level machine code, e.g. in verifying operating systems, embedded code and device drivers. Detailed architecture models have also been used in formalizing multiprocessor memory models, see [15].

In the past, academic work has frequently examined the pedagogical DLX architecture, see [6]. When compared to commercial architectures, DLX is relatively uncomplicated – being a simplified version of the MIPS RISC architecture. Consequently, it is rarely cumbersome to work with the entire DLX instruction set. More recently there has been a move to modelling, and working with, commercial instruction sets (possibly in a reduced form). This has been motivated by a desire to carry out demonstrably realistic case studies, showing that various techniques scale and are not purely "academic" in nature. Common commercial architectures include: IA-32, x86-64, PowerPC, SPARC and ARM. The ARM architecture is ubiquitous in low-powered (mobile and embedded) computing devices – the latest version of the architecture, dubbed ARMv7, is implemented by, for example, the Cortex-A8 processor.

There are many challenges when working with full-blown ISAs, these include: (*i*) official reference manuals are large, stretching to many hundreds of pages – one can easily overlook subtle details or become bogged down with "uninteresting" background information; (*ii*) official descriptions are semi-formal (ambiguous); (*iii*) many details are *implementation dependent* or *unpredictable*; (*iv*)

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 243–258, 2010.

architectures frequently have multiple generations, versions and optional extensions; and (v) large formalizations can stretch the capabilities of interactive theorem provers. Most importantly: how can one be sure that the formalization does not contain bugs? The scale and complexity is such that it is not possible to eradicate all errors by simply eyeballing the specification or examining a few instructions. This paper discusses our experiences with constructing and validating a complete model of the ARMv7 architecture using the HOL4 proof system [16].

2 Approach

Some key aspects of the work presented here are:

- The ARM instruction set architecture has been modelled in HOL using a monadic style. This approach has a number of advantages, which are discussed in Section 3.
- The model is extensive and detailed – it covers all architecture versions currently supported by ARM, including full support for Thumb-2 instructions.[1]
- A collection of tools have been built around the model, making it accessible and easy to work with. This includes an assembler and disassembler, both of which are implemented in Standard ML. There is also a tool for automatically extracting the semantics of a single instruction from the model: this is implemented through evaluation (forward proof) and is discussed in Section 4.
- A distinction is made between entities that are defined or derived inside of the HOL logic and those that reside outside – this is illustrated in Figure 1. It is important that everything defined inside of the logic is valid. On the other hand, although it is advantageous that the other tools (e.g. the parser and encoder) are bug free, these are not fundamentally relied upon in formal verification work.
- The model operates at the *machine code* level, as opposed to a more abstract assembly code level. In particular, the model does not provide assembly level "pseudo instructions" and instruction decoding is explicitly modelled inside the logic. This means that the the model can be directly validated by comparing results against the behaviour of hardware employing ARM processors – this is discussed in Section 5.

Through the use of extensive validation, trust in the model is progressively established. An efficient testing framework has been developed, which is based on a HOL session communicating with an ARM development board (via a serial port). This set-up is required because most PCs are based on x86 processors and cannot natively run ARM code. The results from this testing are discussed in Section 5.3. Due to space constraints, precise details of the ARM architecture are not provided here, readers are instead referred to [4] and [11].

[1] It does not cover the ARMv7-M profile (which has a slightly different programmer's model) or the ThumbEE, Jazelle, VFP and Advanced SIMD extensions.

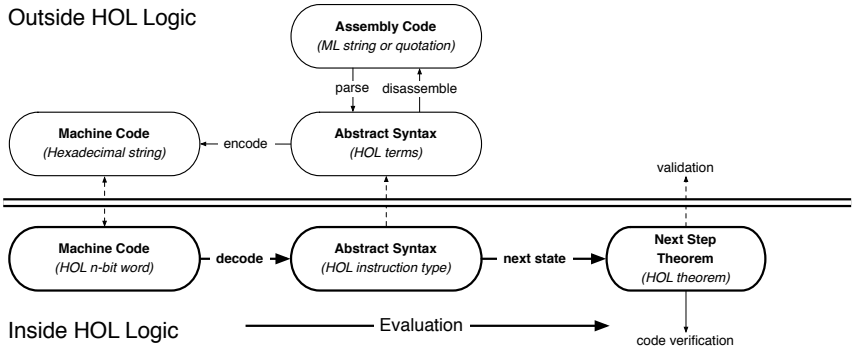


Fig. 1. Overall structure of the formalization

3 Monadic Specification

The HOL4 system provides built-in support for defining recursive, total functions (see [17]). Consequently, formal specifications can be written in a functional programming style using syntax roughly similar to that of ML. For example, in the HOL4 model of the ARMv4 architecture (see [3]), a typical definition is of the following form:

$$\begin{aligned} f(v_1, \ldots, v_m) = \ & \text{let } x_1 = g_1(\ldots) \text{ in} \\ & \ldots \\ & \text{let } x_n = g_n(\ldots) \text{ in} \\ & (w_1, \ldots, w_m) \end{aligned}$$

If f defines the semantics of a machine code instruction then the vector v would represent the components of the *programmer's model* state space (for example, machine registers) and w specifies the next state. The variables x_i are intermediate computations; typically the result of accessing, updating and manipulating state components. There are a few problems with this particular style of specification:

- Explicitly naming state component (splitting the state into a vector) can make it harder to make global changes to sizeable specifications, e.g. adding, removing or changing the type of state components.
- The semantics is rigidly fixed to that of a state transformer. In particular, it is not possible to reason about the order of intermediate computations, observing whether or not they were performed sequentially or in parallel.
- For those more familiar with imperative code, the specification is not especially readable.
- It is not obvious how to handle memory I/O, non-determinism or "error states".

All of these factors motivate the use of a monadic programming style (see [18]), where computations themselves are represented with an abstract data type.

Let $\mathbf{M}$ represent a monad type constructor. The two fundamental monad operations are *return* and *bind*, represented by:

$$\text{return} : \alpha \to \alpha\ \mathbf{M} \quad \text{and} \quad \gg= \ : \alpha\ \mathbf{M} \to (\alpha \to \beta\ \mathbf{M}) \to \beta\ \mathbf{M}$$

respectively. The return operation gives a value to a computation. The bind operation composes computations: it takes the result of one computation and passes it on to another. The type variables α and β represent the types for working values in a computation: these roughly correspond with the variables and arguments of procedures and functions in an imperative language. The monad type constructor $\mathbf{M}$, and associated primitive operations, can be defined in any number of ways, implementing various underlying computational models – this can loosely correspond with defining a semantics (and run-time environment) for a given programming language.

In addition to the two primitive monad operations, our specifications also makes use of a parallel operation:

$$||| : \alpha\ \mathbf{M} \to \beta\ \mathbf{M} \to (\alpha \times \beta)\ \mathbf{M}\ .$$

This performs two operations, but without imposing an order of evaluation.

3.1 Sequential Monad

It is possible to define monads in HOL without considering concrete implementations: one could, for example, provide an axiomatic formalization. However, there are advantages to being able to actually carry out computations with the model (see Section 4). This section presents a *sequential* monad – this has formed the primary basis for working with the ARM specification. The monad provides a simple *operational semantics* in a *shallow embedding* style – this is suited to evaluation and code verification with a Hoare style logic. The overall HOL specification is split into two parts: the monad specification and the instruction set specification. This means that the instruction set specification part can be interpreted by any other monad with the same interface.

The type constructor for the sequential monad is as follows:

$$\alpha\ \mathbf{M_{seq}} \equiv \text{state} \to (\alpha \times \text{state})\ \text{MaybeError}$$

where state is the programmer's model state space and

$$\beta\ \text{MaybeError} = \text{Okay of}\ \beta \mid \text{Error of string}\ .$$

This is essentially a state-transformer monad with error states. The monad type can be viewed as a partial map, representing a state transition with a return value. When the map is "undefined", the result is a tagged string – this can be used to identify where an error occurred.

The basic monad operations are defined as follows:

$$\text{return}\ (v)\ =\ \lambda s.\ \text{Okay}\ (v, s)\ ,$$

$$(f\ \gg\!=\ g) = \lambda s.\ \left[\begin{array}{l}\text{case}\ f(s)\ \text{of Error}\ e \rightarrow \text{Error}\ e \\ \qquad\qquad\quad \mid \text{Okay}\ (v, s') \rightarrow g(v)(s')\end{array}\right]$$

and

$$(f\ \|\|\ g) = (f\ \gg\!=\ (\lambda v_1.\ g\ \gg\!=\ (\lambda v_2.\ \text{return}\ (v_1, v_2))))\ .$$

Note that errors are terminal (no further next state computation is performed) and the parallel operation simply performs computations in a left to right order.

Thanks to Michael Norrish, it is possible to use Haskell's *do-notation* when parsing and printing monadic terms in HOL4. For example, the parallel operation above is more readable when written as follows:

$$(f\ \|\|\ g) = \textbf{do}\ v_1 \leftarrow f;\ v_2 \leftarrow g;\ \text{return}\ (v_1, v_2)\ \textbf{od}\ .$$

In addition to these operations, there is a collection of operations for accessing (reading an writing) state components. For example, registers are accessed with:

$$\text{read\_reg} : \text{iid} \rightarrow \text{bool}[4] \rightarrow \text{bool}[32]\ \mathbf{M}\ \text{and}$$
$$\text{write\_reg} : \text{iid} \rightarrow \text{bool}[4] \rightarrow \text{bool}[32] \rightarrow \text{unit}\ \mathbf{M}$$

where bool[4] is a 4-bit register index (for registers `r0`–`r15`); and bool[32] is a 32-bit register value. The type iid is used to identify threads and is of little interest here.[2] The definitions for these operations derive from pseudo-code contained within the official ARM programmer's model description (see [11]).

3.2 Instruction Set Specification

Having defined the underlying monad, one can then define the semantics of instructions. The following operation runs one instruction:

$$\text{arm\_instr} : \text{iid} \rightarrow \text{encoding} \times \text{bool}[4] \times \text{instruction} \rightarrow \text{unit}\ \mathbf{M}\ .$$

This operation takes a triple (*enc*, *cond*, *ast*), which represents the result of fetching and decoding an instruction. Instructions are conditionally run: for example, the instruction `addcs r1,r2` will have a *cond* value of 2 and it will be a no-op when the carry flag is not set. The *enc* field indicates the instruction encoding, e.g. 16-bit Thumb, 32-bit Thumb or 32-bit ARM. The behaviour of instructions is specified with various sub-operations, these are selected by pattern matching over the abstract syntax term (*ast*).

[2] It allows register and memory accesses to be tagged with the identity of the thread that made them.

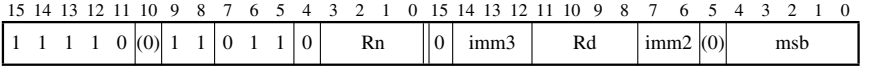
As an example, consider the bit-field-insert instruction (`bfi`).[3] This is specified on page A8-49 of the ARM reference [11] with the following pseudo code:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

The "encoding specific operations" part assigns values to components based on the instruction encoding. For example, with a 32-bit Thumb encoding the following applies from page A8-48:

Encoding T1 ARMv6T2, ARMv7

```
BFI<c> <Rd>,<Rn>,#<lsb>,#<width>
```




```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

The description above additionally provides the concrete syntax and encoding for this instruction, together with a list of architecture versions over which the instruction is defined. The corresponding HOL4 specification for `bfi` is:

```
⊢ bit_field_clear_insert_instr ii enc (Bit_Field_Clear_Insert msb d lsb n) =
  instruction ii "bit_field_clear_insert"
    (thumb2_support CROSS U(:ARMextensions → bool))
    (λ v.
       (if enc = Encoding_Thumb2 then
          ((d = 13w) ∨ (d = 15w)) ∨ (n = 13w)
        else
          d = 15w) ∨ w2n msb < w2n lsb)
    do
      rd ← read_reg ii d |||
      rn ← if n = 15w then return 0w else read_reg ii n;
      increment_pc ii enc |||
      write_reg ii d (bit_field_insert (w2n msb) (w2n lsb) rn rd);
      return ()
    od
```

This code simultaneously specifies the closely related bit-field-clear (`bfc`) instruction. Grouping related instructions together greatly reduces the size specification, which in turn limits the scope for introducing errors. The helper function `instruction` takes: the thread identifier; a string naming the *instruction class* (this is used to tag error states); a set representing the architectures and extensions over which the instruction is defined; a predicate that determines whether the instruction is unpredictable for a particular instruction set version; and an operation that specifies the behaviour when the instruction is defined and predictable. Together with the decoding function, this covers all aspects of the official ARM pseudo code specification.

[3] This replaces a bit range in a destination register with bits from a source register, which is implemented in HOL4 with the bit vector operation `bit_field_insert`.

Although the HOL4 specification is far from being aesthetically perfect, it is at least fairly compact and reasonably readable. More importantly, it is precise and formal. In fact, in order to be of use, the HOL4 specification is in some ways over-precise, since it specifies the order of resource accesses, as well as specifying when the program counter is updated. The ARM reference explicitly states that their pseudo code does not cover such low-level aspects of the behaviour (see page 4 of Appendix I in [11]). However, cases in which such design choices would become visible are typically designated (by ARM) as being unpredictable, so this over-specification should not be a problem at this level of abstraction.

4 Single Step Theorems: Evaluation

For the purposes of code verification and model validation, we require theorems of the form:

$$\vdash \forall s.\ \mathrm{P}(s) \Rightarrow (\mathrm{NEXT}(s) = s') \tag{1}$$

where the predicate P specifies a context (e.g. instruction to be run); and the function NEXT : state → state option defines the next state behaviour for the architecture with respect to the sequential monad. Such *single step* theorems are needed, for example, to generate Hoare triples for every op-code encountered in the machine code of a program being verified. How can such theorems be derived "on-demand" from the monadic specification? First it is necessary to define a monad operation next : unit **M**, which calls `arm_instr` using the result of fetching and decoding an instruction.[4] The following definition is then possible:

$$\mathrm{NEXT}(s) = \left[\begin{array}{l}\text{case next}(s)\ \text{of Error}\ \_ \rightarrow \text{NONE} \\ \qquad\qquad\qquad\quad \mid \text{Okay}\ ((), s') \rightarrow \text{SOME}\ s'\end{array}\right] .$$

It is possible to derive Equation 1 by directly expanding function definitions using the HOL4 simplifier, which supports contextual rewriting.[5] Unfortunately, the simplifier is fairly slow and this is a significant problem when working with such a large model. There is a *much* faster call-by-value rewrite engine (`EVAL`), but this does not directly support contextual rewriting. To get around this limitation, the following theorem (which is proved once and for all) is used:

$$\begin{aligned}\vdash\ & \forall s\ x\ h\ g\ P. \\ & (\forall i.\ P(i) \Rightarrow (g(i) = i)) \wedge && (2)\\ & (\mathrm{next}(g(s)) = \mathrm{Okay}\ ((), x)) \wedge && (3)\\ & (P(s) \Rightarrow (h(g(s)) = x)) && (4)\\ & \Rightarrow \\ & (P(s) \Rightarrow (\mathrm{NEXT}(s) = \mathrm{SOME}\ (h(s)))) \ . && (5)\end{aligned}$$

A tool for deriving single step theorems "on-the-fly" works in stages as follows:

[4] For simplicity sake, non-pipelined operation is assumed here.

[5] Some infrastructure is needed to intelligently expand the context, i.e. generate the predicate P.

- The user supplies an instruction op-code, together with some other context, e.g. the current architecture version, processor mode and instruction set (Thumb or ARM).
- The tool examines the op-code (decodes it) and constructs a custom context predicate P and corresponding function g. Equation 2 is proved to hold. The context predicate and function must ensure that the supplied instruction runs successfully. Amongst other things, this includes avoiding error states, e.g. ensuring that memory addresses are suitably aligned and things like division by zero do not occur. This stage can be completed fairly quickly.
- The next state operation is evaluated for the initial "context" state $g(s)$, giving Equation 3. The rewriter (`EVAL`) is provided with many lemmas to ensure that the evaluation proceeds properly, i.e. making sure that error states are actually avoided. It is also necessary to restrict evaluation from proceeding too far, e.g. expanding with the definition of bit vector operations.
- The terms representing the states $g(s)$ and x are compared and a function h is constructed. Equation 4 is proved to hold.
- The consequent, Equation 5, is derived by *modus ponens* using the general theorem (above) and the three generated theorems (Equations 2–3). This is a simple application of the `MATCH_MP` rule in HOL4. Finally, simplifications are applied and the resulting single step theorem is returned to the user.

Most of the effort with this approach went into automating the construction of P and g, and proving appropriate evaluation lemmas. For example, consider the instruction `eor pc,r1,r2,asr #2`.[6] To be predictable in ARMv7, we require a context P containing $(r_1 \oplus (r_2 \gg 2))[1:0] \neq 2$. This is achieved by defining g such that

$$r_1 \mapsto \textbf{if } (r_1 \oplus (r_2 \gg 2))[1:0] \neq 2 \textbf{ then } r_1 \textbf{ else } r_2 \gg 2$$

and, during the next state evaluation, using the *general* lemma

$$\vdash \forall x\, y.\ ((\textbf{if } (x \oplus y)[1:0] \neq 2 \textbf{ then } x \textbf{ else } y) \oplus y)[1:0] \neq 2$$

which holds because $y \oplus y = 0$ and $0[1:0] \neq 2$. Thus, by applying function g, we successfully satisfy P and avoid the error case – the evaluation proceeds automatically as required. Covering all such cases, over all instructions and architecture versions, was an arduous undertaking. However, the resulting instruction evaluator is relatively fast (see below) and the internal complexities are invisible to the user.

The final simplification stage provides a canonical form for state accesses and updates. For example, registers are read and written using the following functions:

$$\text{ARM\_READ\_REG} : \text{bool}[4] \to \text{state} \to \text{bool}[32] \text{ and}$$
$$\text{ARM\_WRITE\_REG} : \text{bool}[4] \to \text{bool}[32] \to \text{state} \to \text{state}\ .$$

[6] In ARMv7 this instruction performs a *branch with exchange* to the target address $r_1 \oplus (r_2 \gg 2)$, where $\oplus$ is exclusive-or and $\gg$ is signed right shift. The "exchange" part relates to switching between Thumb and ARM code. The behaviour is different for ARMv6 and different again for all earlier versions – the tool is aware of this.

The theorem below is derived from the single step theorem for 32-bit Thumb op-code `F362 01C7` (`bfi r1,r2,#3,#5`):

```
⊢ Abbrev (pc = ARM_READ_REG 15w state) ∧ Abbrev (rd = ARM_READ_REG 1w state) ∧
  Abbrev (rn = ARM_READ_REG 2w state) ⇒
  (ARM_ARCH state = ARMv7_A) ∧ ... ∧ aligned (pc,2) ∧
  (ARM_READ_MEM (pc + 3w) state = 1w) ∧ (ARM_READ_MEM (pc + 2w) state = 199w) ∧
  (ARM_READ_MEM (pc + 1w) state = 243w) ∧ (ARM_READ_MEM pc state = 98w) ⇒
  (ARM_NEXT state = SOME
     (ARM_WRITE_MEM_READ (pc + 3w) (ARM_WRITE_MEM_READ (pc + 2w)
           (ARM_WRITE_MEM_READ (pc + 1w) (ARM_WRITE_MEM_READ pc
                 (ARM_WRITE_REG 1w (bit_field_insert 7 3 rn rd)
```

The first two lines show some *abbreviations* – these have been added here to aid readability. Note that these theorems are not really designed for human consumption – instead, they provide raw input to other automated tools. Observe that memory accesses (from fetching the instruction) have been recorded with `ARM_WRITE_MEM_READ`. This example took around 0.9 s to run,[7] which is approximately the same time that it takes to perform full ground-term evaluation.

5 Validation

Our ARM model formalizes a substantial part of the 2000-page ARM reference manual. As a result, the specification is very large and detailed. The ARM model is sufficiently complex that mistakes are very hard to avoid and very hard to discover. How do we know that our model correctly describes the execution of ARM instructions on ARM processors? Furthermore, if there are mistakes in the model, how do we find them?

Our solution is a validation infrastructure that allows us to compare the execution of ARM instructions in our model with their execution on real ARM hardware. This infrastructure consists of a mixture of ML, C and custom assembly programs, together with the hardware used to run machine code on ARM processors. The following ARM development boards have been used:

Olimex LPC-2129P board, with a comparatively old ARM7TDMI-S core;
Atmel SAM3U-EK board, with a "lightweight" ARM Cortex-M3 core; and
Texas Instruments BeagleBoard, with a "heavyweight" ARM Cortex-A8 core.

The majority of the testing has been performed on the BeagleBoard, which supports the latest architecture version, namely ARMv7-A.

5.1 Random Testing

Our principle validation approach is based on generating large test suites of randomly generated instructions. The generator is designed to provide broad coverage over the ARM and Thumb instruction spaces. The number of instruction instances is sufficiently large that it is not feasible to manually achieve such

[7] For HOL4 (experimental kernel) under Poly/ML with a Pentium 4, 3.0 GHz and 2 GB of RAM.

wide coverage in a reliable fashion. However, in some cases a custom test suit is used, which may include manually selected op-codes. This helps speed up the testing process when examining op-codes that are currently deemed "of interest" or that require "special treatment".

Although substantial software engineering effort went into writing this validation infrastructure, its top-level functionality is conceptually simple:

Step 1: Instruction selection and evaluation.

Instructions are generated by randomly choosing *valid* abstract syntax terms, representing instructions of a given kind, e.g. data-processing, branch, load, store or other. These terms are then encoded into 16-bit or 32-bit instruction op-codes and the ARM model is evaluated for each concrete instruction encoding, i.e. we calculate the step theorem described in Section 4. We ignore instructions for which the model returns "unpredictable".

Step 2: Installing test code onto an ARM board.

With some boards, installation of new programs requires physically removing and inserting jumpers on the boards. (The reason for this is that the boards are implemented as a Harvard architecture, i.e. programs cannot alter themselves or install new code.) Consequently, human interaction is sometimes required, and instructions are generated and tested in batches.

Step 3: Random input generation.

Once the boards have the correct batch of tests installed, test cases can be sent across the serial cable. We generate random inputs for all registers that are, according to the model, relevant for this instruction. In order to increase the chances of hitting corner cases such as "result equals zero", each input is chosen, by a fixed probability, to be one of the following constants:

```
0  1  2  01010101  00FF00FF  FF00FF00  FFFFFFFE  FFFFFFFF
```

otherwise input is chosen uniformly from the set of 32-bit numbers.

Step 4: Sending input via serial cable, waiting for reply.

Input is sent to the boards as strings, e.g. the following echo command will tell the board to test instruction `1917F303` on inputs `20000000 FF00FF00 9E466F33`, if the board is listening to serial port `ttyS0`:

```
echo "1917F303 20000000 FF00FF00 9E466F33" > /dev/ttyS0
```

(To save space, this example omits showing all other register values.) The tester program on the board: reads this input; finds the right instruction to execute; sets up the state; executes the instruction; saves the state and sends back the following output, which can be read from ML as a normal text file at location `/dev/ttyS0`:

```
instruction: 1917F303
input: 20000000 FF00FF00 9E466F33
output: 28000000 FF00FF00 FF800000
```

Programming the board software was by far the hardest part of the validation effort.

Step 5: Validating results against the model.

Once the board has responded to the input, the instruction's step theorem is instantiated and evaluated using the concrete values for the input line (as shown above). If the test results disagree with the model then a failure is reported in a log file, e.g. the following log entry records a genuine error in the first revision of our ARM model. Here the values of the flag register `cpsr` and register `r9` differ from their expected values.

```
FAIL:  1917F303  ARCH ARMv7-M THUMB ssat r9,#24,r3,lsl #4
 resource:   cpsr     r3       r9
 input:      20000000 FF00FF00 9E466F33
 board.out:  28000000 FF00FF00 FF800000
 model.out:  20000000 FF00FF00 00800000
 diff:       ^^^^^^^^          ^^^^^^^^
```

The cause of this error was a minor misinterpretation of the ARM manual.

Step 6: Repeat from Step 3.

By looping through Steps 3–6, we get through five tests per second on average. This speed is achieved by only once evaluating the full ARM model symbolically for each instruction (Step 1) and then in the test loop (Steps 3–6) evaluating only fully instantiated terms, which is relatively fast in HOL. The overall performance is limited by communication speeds.

5.2 Other Means of Gaining Assurance

There are other means of gaining assurance that the model is correct. For example, we gain some assurance that the model cannot be completely wrong from:

- Observing that code verified against this model (see [13]) seems to behave as expected when executed on real hardware.
- Running the model over ARM code that calculates a non-trivial known function, e.g. MD5. For example, a reference C implementation of MD5 (see [14]) was cross-compiled to ARM machine code using GCC. This was then run on an SML version of the HOL model, which was generated using Konrad Slind's EmitML tool. This approach sacrifices trust (using the LCF approach) for performance – running a few thousand ARM instructions per second.

However, both of these approaches are inferior to the testing described above, since these approaches have smaller coverage of the instruction space and make finding the source of erroneous output very complicated.

5.3 Test Results

Comparing the execution of instructions on hardware to evaluations of the ARM model has been a successful method for both quickly finding bugs in the model

and as a means of gaining evidence that the HOL definitions are, if not completely accurate, very close to exactly right. At the time of writing the testing coverage is good but not yet complete. Progress and tests are recorded at www.cl.cam.ac.uk/~mom22/arm-tests. This should allow others to benefit from, and independently assess, this work.

The following bugs were found using the approach described in Section 5.1.

1. Bit-field insert (BFI): the following update should occur
   `Rd<msb:lsb> ← Rn<(msb-lsb):0>`
   but instead the following was occurring
   `Rd<msb:lsb> ← Rn<msb:lsb> .`
2. Signed saturates (SSAT and SSAT16): there was a missing application of a sign-extension function.
3. 16-bit signed saturate (again) and an assortment of signed multiples (SSAT16, SMLA<x><y>, SMUL<x><y>, SMLAW<x><y>, SMULW<x><y> and SMLAL<x><y>): sign extension was not working properly because the bit vector operation word_bits was being used instead of word_extract.
4. The 32-bit Thumb versions of load signed half-word and load signed byte (LDRSH and LDRSB): these were incorrectly decoded (flag values were being extracted from the wrong bit position).

In each of the cases listed above there was a clear discrepancy between the "real" register output values and those obtained through evaluating the model. In addition to these bugs it also became clear that the 32-bit Thumb register shift instructions (LSR, ASR, LSL and ROR) were not being tested. This was because the model was incorrectly identifying them as being unpredictable. It later transpired that there was also a bug in decoding these instruction.[8]

Finally, a bug was found through the MD5 example mentioned in Section 5.2. The condition test was wrong for the greater-than (GT) and less-than or equal (LE) conditions: the carry flag (C) was being used instead of the zero flag (Z).[9]

As an unforeseen consequence of this project, it has has been possible for us to identify and report bugs in the GNU assembler (gas). These mostly concern Thumb-2 support in versions 2.19 and 2.20. That is to say, there were errors in the binary encoding of SEV.W, PKHTB, QADD, QDADD, QSUB and QDSUB. The reported bugs are documented at sourceware.org/bugzilla/ under bug numbers 10168, 10169, 10186 and 11013.

6 Restrictions

There are some limitations to our approach. We have not found a *clean* way to simultaneously consider multiple monadic interpretations of the specification in HOL4. This has not been a problem for our work, where we focus on the sequential semantics, but we speculate that some kind of module system (such as Locales in Isabelle or Sections in Coq) could be helpful here. The testing framework has proved to be very successful, however, note that:

[8] A bit vector extract was off by one position.
[9] This bug was fixed before the random testing covered conditional instructions.

- Store instructions require special treatment.
- Care must be taken with instructions that access or update the program counter or stack pointer (registers fifteen or thirteen). The random instruction generator normally avoids these instructions, since *most* instances are unpredictable. The predictable cases must be tested separately but it is necessary to address the problem of providing a mechanism for safely branching when running tests.
- If something does go wrong (e.g. an op-code is unexpectedly undefined or is a branch) then it can be tricky to recover and work out what has happened. A "hang" must be treated as a possible fail case.
- It is hard to be confident that the coverage is exactly right for each supported architecture version. That is to say, one cannot be totally sure that unpredictable and undefined instruction instances have been properly identified. Testing has not been carried out on ARMv5 or ARMv6 boards. Furthermore, the testing automatically filters out all instructions that the model says are unpredictable and some of these cases are not easy to spot in the ARM reference [11]. Omissions can be spotted by examining the table of results, but this process is not foolproof.
- It is not possible to test instruction instances that need to be run in privileged modes (e.g. supervisor mode) or that change the current processor mode. This affects the testing of `mrs`, `msr`, `cps`, `bkpt`, `rfe`, `svc` and `smc` instructions. This also covers hardware exceptions – interrupts, aborts and resets.
- One cannot fully test implementation dependent or system features. This includes semaphore instructions, such as `ldrex` and `clrex` (clear-exclusive), and hint instructions, such as `wfe`, `wfi` (wait for interrupt), `pld` (pre-load data) and `dmb` (data memory barrier). In some cases it is possible to simply observe whether or not these instruction behave like no-op instructions.

It is possible that many of these shortcomings could be overcome by using the JTAG interface on the development boards, instead of using the serial port. The JTAG interface is specifically designed for carrying out debugging with embedded processors. However, this would require more specialist equipment and know-how. We believe that the testing that has been completed to date provides an excellent basis for establishing trust in the model.

7 Related Work

This section discusses related work in formalizing various *commercial* instruction set architectures using interactive theorem provers, i.e. in ACL2, Coq, Isabelle and HOL4. There is much work that is indirectly related, but here we exclude non-commercial architectures (e.g. DLX) and informal or semi-formal ISA models (e.g. in C, System Verilog, Haskell and so forth). It is worth noting that there have been significant efforts made in testing large formal models in other areas, e.g. network protocols, see [2]. Work in the area of commercial ISAs includes the following.

ARM. The ARM specification presented here has its origins in work on verifying the ARM6 processor to the RTL level, see [3]. The specification of the architecture (then version 3) has been almost completely rewritten in the process of

upgrading to a monadic specification for architecture versions 4–7. Nevertheless, the experience gained from that project was invaluable and it provided an excellent point of reference.

Processor implementations of the modern architecture versions are proprietary and so we are unable to prove our specification correct with respect to RTL level models. Instead we have validated the model through extensive testing against modern ARM hardware.

ARM/C. The L4 verified project [8] has produced a formally verified microkernel running on ARMv6. However, the model stays at and above the C level and only describes how ARM specific details are seen through C code (e.g. details of interrupts). They assume correctness of C compilers and assume the correctness of in-lined ARM assembly, which constitutes approximately 7% of the microkernel's implementation. Their low-level functional specification of the C code uses monads to make it look similar to the original C.

x86. Our work on testing the model against real hardware was inspired by similar work by Susmit et al. [15] on validation of an operational semantics for x86 machine code. We achieved higher throughput of tests by structuring our test framework differently: we evaluated the ARM model once for each concrete instruction instance and reuse the resulting theorem for multiple test of the same instruction, while the x86 work re-evaluated the x86 model for each test and that work did not make use of development boards.

An extensive formal model of the x86 instruction set is being developed by Hunt in conjunction with work on specifying and verifying the media unit, i.e. a unit which performs floating point arithmetic, of a Centaur Technology's x86 processor [7]. As part of this work, Hunt developed the E hardware specification language which has some monad-like features – in so far as allowing the model to support multiple interpretations. Unfortunately this high-fidelity model of the x86 instruction set architecture is not publicly available.

AAMP7G. Another commercial formal specification has been developed by Rockwell Collins. They have an executable ACL2 model of the Rockwell Collins AAMP7G microprocessor at the instruction-set level [5]. Unfortunately, as with Hunt's x86 model, this model is also not in the public domain.

PowerPC. The Compcert project [10] has produced, and proved the correctness of, an optimising C compiler that targets PowerPC. As part of this work they formalized a subset of PowerPC assembly. Their model is smaller in scope than our ARM model (but sufficient for a compiler) and does not include an instruction decoder, thus their model is an assembly level model. They also have a more abstract view of memory which is expressed in terms of memory blocks, in contrast to our very concrete mapping from 32-bit addresses to 8-bit data.

JVM. A succession of increasingly sophisticated models of the JVM bytecode have been developed in ACL2 [12], the most complicated of which includes threaded behaviour and untyped execution. Models of JVM have also been developed in Isabelle/HOL [9] and Coq [1].

8 Summary

The ARMv7 architecture reference [11] is a sizeable document (stretching to over two thousand pages in length) and it covers all aspects of the architecture. This ARM reference has been used to construct a formal instruction set model in HOL using a monadic specification style. In total the specification comes to around 6500 lines of HOL4 script. The model covers many thousands of instruction instances, which perform non-trivial arithmetic and logical bit vector operations. Instruction decoding is modelled explicitly – mapping ARM and Thumb machine code to an AST data type.

Two important questions arise. How to make the model accessible and easy to use in formal verification projects. How to ensure that the model is trustworthy and as free from bugs as possible. To address these points significant tool support has been developed. In fact, this endeavour requires more code than the model itself, accounting for approximately 15000 lines of code/script. The most important tool is an instructions evaluator – this takes an instruction op-code and outputs a theorem giving that instruction's operational semantics. This *single step* theorem can be used in code verification and in validating the model. A novel technique is used to ensure that the evaluator works efficiently and automatically. The formalization is made more accessible through tight integration with a custom written ARM assembler and disassembler. This saves users having to build and rely upon `gas` as a cross-compiler.

The model has been systematically tested through comparison against the behaviour of ARM hardware. Batches of instructions are randomly generated and loaded onto development boards. The single step theorems are used to evaluate the instructions for multiple data inputs (register assignments) and the results are compared against the output from the boards. This technique has enabled us to run many thousands of tests, identifying and fixing a number of bugs in the model. We encourage others to examine and use the model, tools and test data/results, which are publicly available at `www.cl.cam.ac.uk/~acjf3/arm`.

Acknowledgements. Many thanks to all those who have provided valuable support and feedback for this work. In particular, we would like to thank Mike Gordon, Peter Sewell, Scott Owens and Michael Norrish.

References

1. Atkey, R.: CoqJVM: An executable specification of the Java virtual machine using dependent types. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 18–32. Springer, Heidelberg (2008)
2. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 265–276. ACM, New York (2005)
3. Fox, A.: Formal specification and verification of ARM6. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 25–40. Springer, Heidelberg (2003)

4. Furber, S.: ARM: system-on-chip architecture, 2nd edn. Addison-Wesley, Reading (2000)
5. Hardin, D.S., Smith, E.W., Young, W.D.: A robust machine code proof framework for highly secure applications. In: The ACL2 theorem prover and its applications (ACL2 '06), pp. 11–20. ACM, New York (2006)
6. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann, San Francisco (2002)
7. Hunt Jr., W.A., Swords, S.: Centaur technology media unit verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 353–367. Springer, Heidelberg (2009)
8. 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: Symposium on Operating Systems Principles (SOSP), pp. 207–220. ACM, New York (2009)
9. 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)
10. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Principles of Programming Languages (POPL), pp. 42–54. ACM Press, New York (2006)
11. ARM Limited. ARM architecture reference manual: ARMv7-A and ARMv7-R edition. Technical Report ARM DDI 0406B, ARM Limited (2008)
12. Liu, H., Strother Moore, J.: Executable JVM model for analytical reasoning: a study. In: Interpreters, virtual machines and emulators (IVME'03), pp. 15–23. ACM, New York (2003)
13. Myreen, M.O., Gordon, M.J.C.: Verified LISP implementations on ARM, x86 and PowerPC. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 359–374. Springer, Heidelberg (2009)
14. Rivest, R.: The MD5 message-digest algorithm, http://www.ietf.org/rfc/rfc1321.txt (accessed, January 2010)
15. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: Principles of Programming Languages (POPL), ACM, New York (2009)
16. 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)
17. Slind, K.X.: TFL: An environment for terminating functional programs, http://www.cl.cam.ac.uk/~ks121/tfl.html (accessed, January 2010)
18. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science, pp. 61–78 (1992)

Automated Machine-Checked Hybrid System Safety Proofs*

Herman Geuvers[1,2], Adam Koprowski[3],
Dan Synek[1], and Eelis van der Weegen[1]

[1] Radboud University, Nijmegen
[2] Technical University, Eindhoven
[3] MLState, Paris

Abstract. We have developed a hybrid system safety prover, implemented in Coq using the abstraction method introduced by [2]. The development includes: a formalisation of the structure of hybrid systems; a framework for the construction of an abstract system (consisting of decidable "over-estimators" of abstract transitions and initiality) faithfully representing a concrete hybrid system; a translation of abstract systems to graphs, enabling the decision of abstract state reachability using a certified graph reachability algorithm; a proof of the safety of an example hybrid system generated using this tool stack. To produce fully certified safety proofs without relying on floating point computations, the development critically relies on the computable real number implementation of the CoRN library of constructive mathematics formalised in Coq. The development also features a nice interplay between constructive and classical logic via the double negation monad.

1 Introduction

In [2], Alur et al. present an automated method for hybrid system safety verification in which one derives from the hybrid system of interest an *abstract* hybrid system, which is essentially a finite automaton whose traces are sufficiently representative of traces in the original system that unreachability in the abstract system (which can be decided using a standard graph algorithm) implies unreachability in the concrete system (which, being governed by continuous behaviours, cannot be decided so readily). Thus, the abstraction method brings the safety verification problem from a continuous and infinite domain into a discrete and finite domain, where it can be dealt with using standard graph algorithms.

The prototype implementation described in [2] was developed in a conventional programming language, only has an informal correctness argument, and uses ordinary floating point numbers to approximate the real numbers that are used in said argument. These factors limit the confidence one can justifiably have in safety judgements computed by this implementation, because (1) it is easy

* This research was supported by the BRICKS/FOCUS project 642.000.501, Advancing the Real use of Proof Assistants.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 259–274, 2010.

for bugs to creep into uncertified programs; (2) it is easy to make mistakes in informal correctness arguments; and (3) floating point computations are subject to rounding errors and representation artifacts.

Our goal is to increase this degree of confidence by developing a *certified* reimplementation of the abstraction technique in Coq, a proof assistant based on a rich type theory that also functions as a (purely functional) programming language. The Coq system lets us develop the algorithms and their formal correctness proofs in tandem in a unified environment, addressing (1) and (2) above.

To address (3), we replace the floating point numbers with exact computable reals, using the certified exact real arithmetic library developed by O'Connor [14] for CoRN, our Coq repository of formalised constructive mathematics [7]. This change is much more than a simple change of representation, however; because computable reals only permit observation of arbitrarily close approximations, certain key operations on them (namely naive comparisons) are not decidable. The consequences of this manifest themselves in our development in several ways, which we discuss in some detail. Hence, our development also serves to showcase O'Connor's certified exact real arithmetic library applied to a concrete and practical problem.

On a separate note, we argue that the use of computable reals is not just a pragmatic choice necessitated by the need to compute, but is actually fundamental considering their role in hybrid systems, where they represent physical quantities acted upon by a device with sensors and actuators. In the real world, measurements are approximate.

The end result of our work is a framework with which one can specify (inside Coq) a concrete hybrid system, set some abstraction parameters, derive an abstract system, and use it to compute (either inside Coq itself or via extraction to OCaml) a safety proof for the concrete system.

2 Hybrid Systems and the Abstraction Method

A hybrid system is a model of how a software system (running on a device with sensors and actuators), described as a finite set of *locations* with (discrete) transitions between them, acts on and responds to a set of continuous variables (called the *continuous state space*), typically representing physical properties of some environment (such as temperature and pressure).

There are many varieties of hybrid systems [9,11]. We follow [2] and to illustrate the definition and the abstraction method, we use the example of a system describing a thermostat (this is the same example as in [2]), shown in Figure 1.

The thermostat has three locations. The first two, Heat and Cool, represent states in which the thermostat heats and cools the environment it operates in, respectively. The third, Check, represents a self-diagnostic state in which the thermostat does not heat or cool. The continuous state space of the thermostat consists of two continuous variables denoting an internally resettable clock c and the temperature T in the environment in which the thermostat operates.

Each location has an associated *invariant* predicate defining the set of permitted values for the continuous variables while in that location. The invariants for the thermostat are:

$$\mathrm{Inv}_{\mathsf{Heat}}(c,T) := T \leqslant 10 \wedge c \leqslant 3, \quad \mathrm{Inv}_{\mathsf{Cool}}(c,T) := T \geqslant 5, \quad \mathrm{Inv}_{\mathsf{Check}}(c,T) := c \leqslant 1.$$

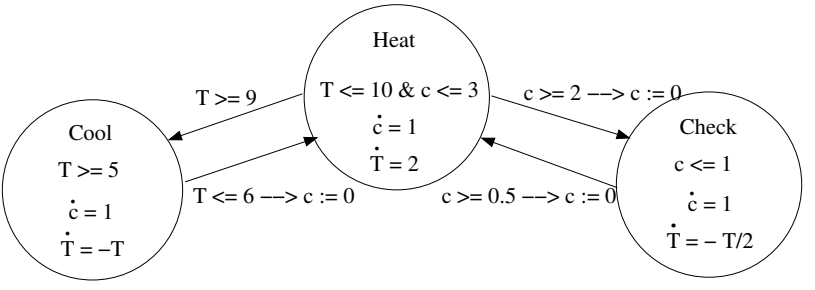


Fig. 1. The Thermostat as an example of a Hybrid Systems

The *initial states* of a hybrid system are determined by a predicate Init. For the thermostat, $\mathrm{Init}(l,c,T)$ is defined as $l = \mathsf{Heat} \wedge c = 0 \wedge 5 \leqslant T \leqslant 10$.

The *discrete* transitions between locations describe the logic of the software system. Each such transition is comprised of two components: a *guard* predicate defining a subset of the continuous state space in which the transition is enabled (permitted), and a *reset* function describing an instantaneous change applied as a side effect of the transition, as seen in the following definition of the discrete transition relation:

$$(l,p) \rightarrow_D (l',p') := \mathrm{guard}_{l,l'}(p) \wedge \mathrm{reset}_{l,l'}(p) = p' \wedge \mathrm{Inv}_l(p) \wedge \mathrm{Inv}_{l'}(p')$$

It will be clear from Figure 1 what the guards and reset functions are. Note the inherent non-determinism in a Hybrid Systems specification: when in Cool, the system can jump to Heat whenever the temperature T is in the interval $[5,6]$.

Each location in a hybrid system has an accompanying *flow function* which describes how the continuous variables change over time while the system is in that location. The idea is that different locations correspond to different uses of actuators available to the software system, the effects of which are reflected in the flow function. In the thermostat example, the flow function corresponding to the Cool location has the temperature decrease over time. This is expressed via the differential equation $\dot{T} = -T$, which is the usual short hand for $T'(t) = -T(t)$, where $T'(t)$ denotes the derivative of the temperature function over time t.

In the canonical definition of hybrid systems, flow functions are specified as solutions to differential equations (or differential inclusions) describing the dynamics of the continuous variables. We follow [2] in abstracting from these, taking instead the solutions of these differential equations, which are flow functions Φ which satisfy:

$$\Phi(p,0) = p \quad \text{and} \quad \Phi(p,d+d') = \Phi(\Phi(p,d),d')$$

The idea is that $\Phi(p,d)$ denotes the value of the continuous variable after duration d, starting from the value p. We say that there is a (concrete) *continuous transition* from a state (l,p) to a state (l,p') if there is a non-negative duration d such that $p' = \Phi_l(p,d)$ with the invariant for l holding at every point along the way:

$$(l,p) \to_C (l,p') := \exists_{d \in \mathbf{R}_{\geqslant 0}} . \Phi_l(p,d) = p' \wedge \forall_{0 \leqslant t \leqslant d} . \mathrm{Inv}_l(\Phi_l(p,t)).$$

A flow function on $\mathbb{R}^2$ can be expressed as the product of two flow functions: $\Phi_l((c_0,T_0),t) = (\varphi_{l,\mathsf{c}}((c_0,T_0),t), \varphi_{l,\mathsf{T}}((c_0,T_0),t))$. In the thermostat example, as in many other examples of hybrid systems, $\varphi_{l,\mathsf{c}}((c_0,T_0),t)$ does not actually depend on T_0 and $\varphi_{l,\mathsf{T}}((c_0,T_0),t)$ does not actually depend on c_0. We call this feature *separability* of the flow function. Our development currently relies heavily on this property. Separability makes the form of the flow functions simpler:

$$\Phi_l((c_0,T_0),t) = (\varphi_{l,\mathsf{c}}(c_0,t), \varphi_{l,\mathsf{T}}(T_0,t))$$

In the thermostat, $\varphi_{l,\mathsf{c}}(c_0,t) = c_0 + t$ for all locations l, $\varphi_{\mathsf{Heat},\mathsf{T}}(T_0,t) = T_0 + 2t$, $\varphi_{\mathsf{Check},\mathsf{T}}(T_0,t) = T_0 * e^{-\frac{1}{2}t}$ and $\varphi_{\mathsf{Cool},\mathsf{T}}(T_0,t) = T_0 * e^{-t}$. So $\varphi'_{\mathsf{Cool},\mathsf{T}}(T_0,t) = -\varphi_{\mathsf{Cool},\mathsf{T}}(T_0,t)$, solving the differential equation $\dot{T} = -T$ for the Cool location.

A transition is either continuous or discrete: $\to_{CD} := \to_D \cup \to_C$. A finite sequence of transitions constitutes a *trace* and we denote by $\twoheadrightarrow_{CD}$ the transitive reflexive closure of $\to_{CD}$. We now say that a state s is *reachable* if there is an initial state i from which there is a trace to s, that is

$$\mathrm{Reach}(s) := \exists_{i \in \mathrm{State}} . \mathrm{Init}(i) \wedge i \twoheadrightarrow_{CD} s.$$

The objective of hybrid system safety verification is to show that the set of reachable states is a subset of a predefined set of "safe" states. For the thermostat, the intent is to keep the temperature above 4.5 degrees at all times, and so we define $\mathrm{Safe}(c,T) := T > 4.5$ (and $\mathrm{Unsafe}(c,T)$ as its complement).

2.1 The Abstraction Method

There are uncountably many traces in a hybrid system, so safety is undecidable in general. In concrete cases, however, safety may be (easily) provable if one finds the proper *proof invariant*. Unfortunately these are often hard to find, so we prefer methods that are more easily automated. The *predicate abstraction* method of [2] is one such method.

The idea is to divide the continuous state space into a finite number of convex subsets (polygons), $A_1, \ldots, A_n$, which yields a finite *abstract state space*, $\mathrm{AState} := \{(l,A_i) \mid l \in \mathrm{Loc}, 1 \leqslant i \leqslant n\}$, with an obvious embedding $A : \mathrm{State} \to \mathrm{AState}$ of concrete states into abstract states. On this abstract state space, one immediately defines *abstract continuous transitions* and *abstract discrete transitions* (both potentially undecidable) as follows.

$$(l,P) \overset{A}{\to}_C (l,Q) := \exists_{p \in P, q \in Q} . (l,p) \to_C (l,q)$$
$$(l,P) \overset{A}{\to}_D (l',Q) := \exists_{p \in P, q \in Q} . (l,p) \to_D (l,q).$$

Define *abstract reachability* by $\mathrm{AReach}(a) := \exists_{s_0 \in \mathrm{State}} . \mathrm{Init}(s_0) \wedge A(s_0) \overset{A}{\rightarrow\!\!\!\!\rightarrow}_{CD} a$, as expected. Also the predicates ASafe and AUnsafe, stating when abstract states are safe / unsafe can be defined in the straightforward way.

Traces in the finite transition system constructed in this way are sufficiently representative (see Figure 2) of those in the original (concrete) system that one can conclude safety of the latter from safety of the abstract system:

$$\text{if } \forall_{a \in \mathrm{AState}} . \mathrm{AReach}(a) \rightarrow \mathrm{ASafe}(a), \text{then } \forall_{s \in \mathrm{State}} . \mathrm{Reach}(s) \rightarrow \mathrm{Safe}(s).$$

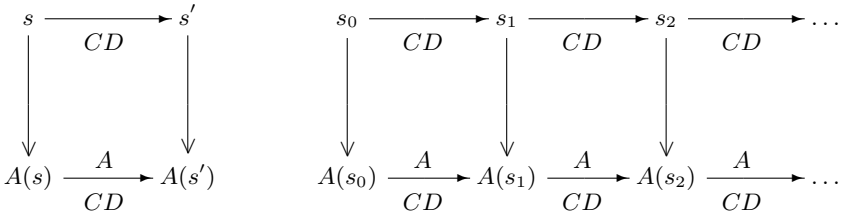


Fig. 2. The abstraction function

The interest and power of the abstraction method lies in two facts. First, we do not need the *exact* definitions of $\overset{A}{\rightarrow}_C$ and $\overset{A}{\rightarrow}_D$ to conclude safety of the concrete system from safety of the abstract system. We only need the property of Figure 2, so we can *over-estimate* $\overset{A}{\rightarrow}_C$ and $\overset{A}{\rightarrow}_D$ (i.e. replace it with a transition relation that allows more transitions). Second, there are good heuristics for how to divide the continuous state space into regions, and how to decide whether there should be an abstract transition from one abstract state to another.

This is indicated in Figure 3. The left hand side illustrates the challenge: given abstract regions A and B, we are to determine whether some flow duration permits flow from points in A to points in B. Following the over-estimation property just mentioned, we introduce an abstract transition from A to B whenever we cannot positively rule this out.

On the right hand side we see the abstract state space indicated for the location Heat. The abstract state space consists of rectangles, possibly degenerate (extending to $-\infty$ or $+\infty$). According to [2], a good candidate for an abstraction is to take the values occurring in the specification (Figure 1) as the bounds of such rectangles. (In case one cannot prove safety, there is of course the opportunity for the user to refine the bounds.) The grey area indicates that from these states also abstract discrete transitions are possible. The dashed area is unreachable, because of the invariant for the Heat location. being reachable. All the abstract transitions from the rectangle $[0.5, 1) \times [5, 6)$ are shown: as the temperature flow function for Heat is $\varphi_{\mathsf{Heat},\mathsf{T}}(T_0, t) = T_0 + 2 * t$, and the clock flow function is $\varphi_{\mathsf{Heat},\mathsf{c}}(c_0, t) = c_0 + t$, these are all the possible abstract transitions.

Using the abstraction method, [2] proves the correctness of the thermostat.

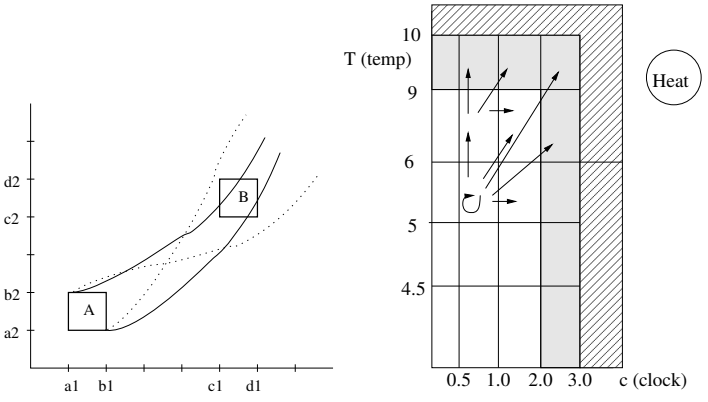


Fig. 3. The abstraction function computed

3 Formalisation

We now describe the Coq formalisation and the design choices made. We will not pay much attention to the specifics of Coq and its type theory CiC, and will instead focus on concerns relating to the use of computable reals and constructive logic. The complete sources of the development are available on the web, as well as a technical report describing the formalisation in more detail [22].

3.1 (Concrete) Hybrid Systems

We begin by showing our definition of a concrete system, the different parts of which we discuss in the remainder of this section.

Record *System* : *Type* :=
$\{$*Point* : *CSetoid*
; *Location* : *Set*
; *Location_eq_dec* : *EqDec Location eq*
; *locations* : *ExhaustiveList Location*
; *State* := *Location* × *Point*
; *initial* : *State* → *Prop*
; *invariant* : *State* → *Prop*
; *invariant_initial* : *initial subsetof invariant*
; *invariant_stable* : $\forall$ *s*, *Stable* (*invariant s*)
; *flow* : *Location* → *Flow Point*
; *guard* : *State* → *Location* → *Prop*
; *reset* : *Location* → *Location* → *Point* → *Point*$\}$.

This is a Coq definition of a record type of "Systems", which contain a field "Point", representing the continuous state space and a field "Location", representing the set of locations. Here, we take Point to be an arbitrary constructive

setoid, which is basically just a type with an equality on it. For Location, we assume a decidable equality and a finite enumeration ("exhaustive list") of locations. The other parts are as expected ("Flow Point" is the type of flow functions on the type "Point"), except for the requirement that the invariant Inv_l is "Stable" for every location l, which we will discuss now.

3.2 Stability, Double Negation, and Computable Reals

Constructively, a proof of $X \to A \vee B$ is a function that, given an X, returns either a proof of A, or a proof of B. With this in mind, suppose we try to implement $le\_lt\_dec : \forall\ (x, y : CR), (x \leqslant y \vee y < x)$, where CR are the constructive reals. Then given x and y in CR, we are to produce a proof either of $x \leqslant y$ or of $y < x$. Unfortunately, the nature of computable reals only lets us observe arbitrarily close approximations of x and y. If it happens to be the case that $x = y$, then no matter how closely we approximate x and y, the error margins (however small) will always leave open the possibility that y is really smaller than x. Consequently, we will never be able to definitively conclude that $x \leqslant y$.

Computable reals do admit two variations of the proposition:

1. $leltdec_{overlap} : \forall\ (x\ y : CR), (x < y \to \forall\ z, (z \leqslant y \vee x \leqslant z))$
2. $leltdec_{DN} : \forall\ (x\ y : CR), \neg\neg(x \leqslant y \vee y < x)$

Both are weaker than the original, and are less straightforward to use. Nevertheless, this is the path we will take in our development (we will heavily use $leltdec_{DN}$), because just taking le_lt_dec as an axiom amounts to cheating. A question that immediately arises is: How does one actually use this doubly negated variant in proofs? One practical way is to observe that double negation, as a function on propositions, is a monad [21]. Writing $DN\ P$ for $\neg\neg P$, we have the following two key operations that make DN a monad:

$$return_{DN} : \forall\ A, A \to DN\ A$$
$$bind_{DN} : \forall\ A\ B, DN\ A \to (A \to DN\ B) \to DN\ B$$

The first expresses that any previously obtained result can always be inserted "into" the monad. The second expresses that results inside the monad may be used freely in proofs of additional properties in the monad. For instance, one may $bind_{DN}$ a proof of $DN(x \leqslant y \vee y < x)$ (obtained from $leltdec_{DN}$ above) with a proof of $(x \leqslant y \vee y < x) \to DNP$, yielding a proof of $DN\ P$.

Thus, DN establishes a "proving context" in which one may make use of lemmas yielding results inside DN that may not hold outside of it (such as $leltdec_{DN}$), as well as lemmas yielding results not in DN, which can always be injected into DN using $return_{DN}$. The catch is that such proofs always end up with results in DN, which begs the question: what good is any of this? In particular, can $leltdec_{DN}$ be used to prove anything not doubly negated?

As it happens, some propositions are *stable* in the sense that they are constructively equivalent to their own double negation. Examples include negations, non-strict inequalities on real numbers, and any decidable proposition.

Requiring that hybrid system invariants are stable effectively lets us use classical reasoning when showing that invariants hold in certain states. One instance where we need this is in the proof of transitivity of the concrete continuous transition.

Invariants are typically conjunctions of inequalities, which are stable only if the inequalities are non-strict. Hence, the limits on observability of computable real numbers ultimately mean that our development cannot cope with hybrid systems whose location invariants use strict inequalities. We feel that this is not a terrible loss. In Section 3.3 we will see analogous limitations in the choice of abstraction parameters.

3.3 Abstract Hybrid Systems

We now want to define an abstract system and an abstraction function satisfying the properties indicated in Figure 2. However, this is not possible, because we cannot make a case distinction like $x \leqslant 0 \vee 0 < x$ and therefore we cannot define a function that maps a point (c, T) to the rectangle R it is in. We can define a function that approximates a point (c, T) up to, say ϵ ($\epsilon > 0$) and then decides to send that point to the rectangle R the approximation is in. This implies that, when one is close to the edge of a rectangle,

- different *representations* of a point (c, T) may be sent to different rectangles,
- a point that is less than ϵ outside the rectangle R may still be sent to R.

The second is very problematic, because it means the property for the abstraction function A depicted in Figure 2 no longer holds.

We argue that these problems are not merely inconvenient byproducts of our use of constructive logic and computable reals, but actually reflect the profound limitation of physical reality where one can only ever measure quantities approximately, making case distinctions like $x \leqslant 0 \vee 0 < x$ simply unrealistic.

Moreover, we claim that the classical abstraction method allows one to prove the safety of systems that are unreliable in practice. We will not expand on this here, but suppose we add a fourth location Off to the thermostat of Figure 1, with $\dot{T} = -1$, $\dot{c} = 1$ and an arrow from Heat to Off with guard $c \geqslant 2 \wedge T < 9$. Clearly, if the system can end up in location Off, it is unsafe. Now, using the classical abstraction method, there is no transition to any state involving location Off from the initial state, because as soon as $c \geqslant 2$, $T \geqslant 9$. However, when we get close to $c = 2$, any small mistake in the measurement of T may send the system to Off, making the whole hybrid system very unreliable.

The positive thing is that we do not really need the commutation property of Figure 2. To address the problems we

- let regions in the abstract hybrid systems overlap (ideally as little as possible, e.g. only at the edges).
- replace the abstract relations $\overset{A}{\rightarrow}_C$ and $\overset{A}{\rightarrow}_D$ by functions over_cont_trans and over_disc_trans that take a region R_0 as input and output a *list of regions*

including R_0: $(R_o, R_1, \ldots, R_n)$ in such a way that $\cup_{0 \leqslant i \leqslant n} R_i$ is an over-approximation of the set of states reachable by a continuous (resp. discrete) step from a state in R_0.
– loosen the requirement on the abstraction function A; for $s \in \text{State}$, we only require $DN(\exists_{r \in \text{Region}} . s \in r)$.

To summarise, if $s \rightarrow_C s'$, then we don't require $A(s')$ to be in the list over_cont_trans$(A(s))$, but we only require s' to be in the $\bigcup$ over_cont_trans$(A(s))$. This simple change relieves us from having to determine the exact regions that points are in: they just should be covered. The functions over_cont_trans and over_disc_trans yield a notion of *trace* in the abstract hybrid system in the straightforward way: starting from R_0, take an R_1 in over_cont_trans(R_0), then an R_2 in over_disc_trans(R_1), and so forth.

Whereas in a concrete hybrid system states consist of a location paired with a point in the continuous state space, in an abstract hybrid system states consist of a location paired with the "name" of a region corresponding to a subset of the continuous state space. From now on we will use a "*concrete*." prefix for names like *State* defined in section 3.1, which now have abstract counterparts. *Region* is a field from a record type *Space* bundling region sets with related requirements:

Record *Space* : *Type* :=
{*Region* : *Set*
; *Region_eq_dec* : *EqDec Region eq*
; *regions* : *ExhaustiveList Region*
; *NoDup_regions* : *NoDup regions*
; *in_region* : *Container Point Region*
; *regions_cover* : $\forall$ (l : *Location*) (p : *Point*),
invariant $(l, p) \rightarrow DN$ {r : *Region* | $p \in r$}}.

The *Container Point Region* type specified for *in_region* reduces to *Point* → *Region* → *Prop*. *Container* is a type class that provides the notation "$x \in y$". Finally, *regions_cover* expresses that each concrete point belonging to a valid state must be represented by a region—a crucial ingredient when arguing that unreachability in the abstract system implies unreachability in the concrete system. The double negation in its result type is both necessary and sufficient:

It is *necessary* because *regions_cover* boils down to a (partial) function that, given a concrete point, must select an abstract region containing that point. This means that it must be able to decide on which side of a border between two regions the given point lies. As we saw in section 3.2, that kind of decidability is only available inside *DN* unless all region borders have nontrivial overlap, which is undesirable.

Fortunately, the double negation is also *sufficient*, because we will ultimately only use *regions_cover* in a proof of ... → ¬*concrete.reachable s* (for some universally quantified variable s), which, due to its head type being a negation, is stable, and can therefore be proved in and then extracted from *DN*. Hence, we only need *regions_cover*'s result in *DN*.

3.4 Under-Estimation and Over-Estimation

Ultimately, in our development we are writing a program that *attempts* to produce hybrid system safety proofs. Importantly, we are *not* writing a complete hybrid system safety decision procedure: if the concrete system is unsafe or the abstraction method fails, our program will simply not produce a safety proof. It might seem, then, that we are basically writing a *tactic* for a particular problem domain. However, tactics in Coq are normally written in a language called Ltac, and typically rely on things like pattern matching on syntax. Our development, on the other hand, is very much written in regular Gallina, with hardly any significant use of Ltac.

We define *underestimation* P to be either a proof of P, or not:

$$\textbf{Definition}\ \mathit{underestimation}\ (P : \mathit{Prop}) : \mathit{Set} := \{b : \mathit{bool} \mid b = \mathit{true} \to P\}.$$

The *bool* in the definition nicely illustrates why we call this an "under-estimation": it may be *false* even when P holds. We can now describe the functionality of our program by saying that it under-estimates hybrid system safety, yielding a term of type *underestimation Safe*, where *Safe* is a proposition expressing safety of a hybrid system.

Considered as theorems, under-estimations are not very interesting, because they can be trivially "proved" by taking *false*. Hence, the value of our program is not witnessed by the mere fact that it manages to produce terms of type *underestimation Safe*, but rather by the fact that when run, it actually manages to return *true* for the hybrid system we are interested in (e.g. the thermostat). It is for this reason that we primarily think of the development as a program rather than a proof, even though the program's purpose is to produce proofs.

The opposite of an under-estimation is an over-estimation:

$$\textbf{Definition}\ \mathit{overestimation}\ (P : \mathit{Prop}) : \mathit{Set} := \{b : \mathit{bool} \mid b = \mathit{false} \to \neg P\}.$$

Since hybrid system safety is defined as unreachability of unsafe states, we may equivalently express the functionality of our development by saying that it over-estimates unsafe state reachability. Indeed, most subroutines in our programs will be over-estimators rather than under-estimators. Notions of over-estimation and under-estimation trickle down through all layers of our development, down to basic arithmetic. For instance, we employ functions such as (recall that *CR* denotes the type of constructive reals):

$$\mathit{overestimate}_{\leqslant_{CR}}\ (\epsilon : \mathbb{Q}^+) : \forall\ x\ y : \mathit{CR}, \mathit{overestimation}\ (x \leqslant_{CR} y)$$

As discussed earlier, $\leqslant_{CR}$ is not decidable. $\mathit{overestimate}_{\leqslant_{CR}}$ merely makes a "best effort" to prove $\neg(x \leqslant_{CR} y)$ using ϵ-approximations. A smaller ϵ will result in fewer spurious *true* results.

3.5 Abstract Space Construction

When building an abstract system, one is in principle free to divide the continuous state space up whichever way one likes. However, if the regions are too

fine-grained, there will have to be very many of them to cover the continuous state space of the concrete system, resulting in poor performance. On the other hand, if the regions are too coarse, they will fail to capture the subtleties of the hybrid system that allow to prove it safe (if indeed it is safe at all). Furthermore, careless use of region overlap can result in undesirable abstract transitions (and therefore traces), adversely affecting the abstract system's utility.

In [2], a heuristic for interval bound selection is described, where the bounds are taken from the constants that occur in the invariant, guard, and safety predicates. For the thermostat, we initially attempted to follow this heuristic and use the same bounds, but found that due to our use of computable reals, we had to tweak the bounds somewhat to let the system successfully produce a safety proof. Having to do this "tweaking" manually is clearly not ideal. One may want to develop heuristics for this.

Another way in which our thermostat regions differ from [2] lies in the fact that our bounds are always inclusive, which means adjacent regions overlap in lines.

3.6 Abstract Transitions and Reachability

Once we have a satisfactory abstract *Space*, our goal is to construct an over-estimatable notion of abstract reachability implied by concrete reachability, so that concrete unreachability results may be obtained simply by executing the abstract reachability over-estimator. We first over-estimate the continuous transitions; we need the following definition for that.

Definition *shared_cover*
$(cs : concrete.State \to Prop)\ (ss : abstract.State \to Prop) : Prop :=$
$\forall\ s : concrete.State, s \in cs \to DN\ (\exists\ r : abstract.State, s \in r \wedge r \in ss).$

A set of concrete states is said to be sharedly-covered by a set of abstract states if for each of the concrete states in the former there is an abstract state in the latter that contains it.

We now specify what the type of *over_cont_trans* should be.

$over\_cont\_trans : \forall\ s : abstract.State,$
$\{p : list\ abstract.State \mid NoDup\ p \wedge shared\_cover$
$(concrete.invariant \cap (overlap\ s \circ flip\ concrete.cont\_trans))\ p\}$

So, *over_cont_trans* s should produce a list of abstract states p without duplicates, such that p is a shared cover of the collection of concrete states c that satisfy the invariant and whose set of origins under *concrete.cont_trans* have an overlap with s. In more mathematical terms: p should form a shared cover of $\{c \in \text{State} \mid \text{Inv}(c) \wedge s \cap \{c' \mid c' \to_C c\} \neq \emptyset\}$. We similarly specify *over_disc_trans* as an over-estimator for *concrete_disc_trans* and *over_initial* as an over-estimator of *concrete.initial*.

We now consider the properties we require for *abstract.reachable*. An obvious candidate is:

$$\forall\,(s : \mathit{concrete.State}), \mathit{concrete.reachable}\ s \rightarrow \\ \forall\,(s' : \mathit{abstract.State}), s \in s' \rightarrow \mathit{abstract.reachable}\ s'.$$

because it implies

$$\forall\,(s : \mathit{concrete.State}) \\ (\exists\, s' : \mathit{abstract.State}, s \in s' \wedge \neg \mathit{abstract.reachable}\ s') \rightarrow \\ \neg \mathit{concrete.reachable}\ s,$$

This expresses that to conclude unreachability of a concrete state, one only needs to establish unreachability of *one* abstract state that contains it. However, this definition neglects to facilitate *sharing*: a concrete state may be in more than one abstract state. So, if a concrete state is in one abstract state which is unreachable, it may still be in another abstract state which *is* reachable. One should establish unreachability of *all* abstract states containing the concrete state. Hence, what we really want is an *abstract.reachable* satisfying:

$$\forall\, s : \mathit{concrete.State}, \\ (\forall\, s' : \mathit{abstract.State}, s \in s' \rightarrow \neg \mathit{abstract.reachable}\ s') \rightarrow \\ \neg \mathit{concrete.reachable}\ s.$$

3.7 Under-Estimating Safety

We now show how a decision procedure for *abstract.reachable* lets us under-estimate hybrid system safety, and in particular, lets us obtain a proof of thermostat safety. (The construction of the decision procedure itself is detailed in the next section.) So suppose we have $\mathit{reachable\_dec} : \mathit{decider}\ \mathit{abstract.reachable}$. *ThermoSafe* is defined as $\mathit{thermo\_unsafe} \subseteq \mathit{concrete.unreachable}$. Since we trivially have $\neg \mathit{overlap}\ \mathit{unsafe}\ \mathit{concrete.reachable} \rightarrow \mathit{ThermoSafe}$, we also have:

Definition $\mathit{under\_thermo\_unsafe\_unreachable} : \mathit{underestimation}\ \mathit{ThermoSafe}$.

Using a tiny utility *underestimation_true* of type $\forall\ P\ (o{:}\mathit{underestimation}\ P), o = \mathit{true} \rightarrow P$, we can now *run* this under-estimator to obtain a proof of the thermostat system's safety:

Theorem : *ThermoSafe*.
Proof.
apply (*underestimation_true under_thermo_unsafe_unreachable*).
vm_compute.reflexivity.
Qed.

The first *apply* reduces the goal to

$$\mathit{under\_thermo\_unsafe\_unreachable} = \mathit{true}.$$

The *vm_compute* tactic invocation then forces evaluation of the left hand side, which will in turn evaluate *over_thermo_unsafe_reachable*, which will evaluate *reachable_dec*, which will evaluate the over-estimators of the continuous and discrete transitions. This process, which takes about 35 seconds on a modern

desktop machine, eventually reduces *under_thermo_unsafe_unreachable* to *true*, leaving $true = true$, proved by *reflexivity*.

We can now also clearly see what happens when the abstraction method "fails" due to poor region selection, overly simplistic transition/initiality over-estimators, or plain unsafety of the system. In all these cases, *vm_compute* reduces *under_thermo_unsafe_unreachable* to *false*, and the subsequent *reflexivity* invocation will fail.

This concludes the high level story of our development. What remains are the implementation of *reachable_dec* in terms of the decidable over-estimators for abstract initiality and continuous and discrete transitions, and the implementation of those over-estimators themselves. The former is a formally verified graph reachability algorithm, that we don't detail here. The over-estimator for continuous transitions, *over_cont_trans* will be detailed in the next section for the thermostat case.

3.8 Over-Estimating Continuous Abstract Transitions

We now discuss the implementation of *over_cont_trans*. Given two regions *r_src* and *r_dst*, if we can determine that there are no points in *r_src* which the flow function maps to points in *r_dst*, then we don't put an abstract continuous transition between *r_src* and *r_dst*. Clearly, this is impossible to meaningfully over-estimate for a general flow function and general regions. However, the thermostat posesses three key properties that we can exploit:

1. its continuous space is of the form $\mathbf{R}^n$;
2. abstract regions correspond to multiplied $\mathbf{R}$ intervals;
3. its flow functions are both *separable* and *range invertible*.

The notion of *separability* has already been discussed in Section 2.
A flow function f on CR is *range invertible* if

$$\exists\ (range\_inverse : OpenRange \to OpenRange \to OpenRange),$$
$$\forall\ (a : OpenRange)\ (p : CR), p \in a \to$$
$$\forall\ (b : OpenRange)\ (d : Duration), f\ p\ d \in b \to d \in range\_inverse\ a\ b$$

Here, *OpenRange* represents potentially unbounded intervals in $\mathbf{R}$ (with bounds closed if present. In other words, if $\varphi : \mathbf{R}^2 \to \mathbf{R}$ is a flow function with range inverse F and a, b are intervals in $\mathbf{R}$, then $F(a, b)$ is an interval that contains all t for which $\varphi(x, t) \in b$ for some $x \in a$. Range invertibility is a less demanding alternative to point invertibility: φ^{-1} is the *point inverse* of φ if $\forall x, y \in \mathbf{R}(\varphi(x, \varphi^{-}1(x, y)) = y)$. So a point inverse $\varphi^{-1}(x, y)$ computes the exact time t it takes to go from x to y via flow φ. A range inverse computes an interval that contains this t.

In the formalisation we use a modest library of flow functions when defining the thermostat's flow. Included in that library are range-inverses, which consequently automatically apply to the thermostat's flow. Hence, no ad-hoc work is needed to show that the thermostat's flow functions are range-invertible. Having

defined the class of separable range-invertible flow functions, and having argued that the thermostat's flow is in this class, we now show how to proceed with our over-estimation of existence of points in r_src which the flow function map to points in r_dst. Regions in the abstract space for our thermostat are basically pairs of regions in the composite spaces, so r_src and r_dst can be written as $(r\_src\_temp, r\_src\_clock)$ and $(r\_dst\_temp, r\_dst\_clock)$, respectively, where each of these four components are intervals.

We now simply use an *OpenRange* overlap over-estimator of type

$$\mathbb{Q}^+ \to \forall\ a\ b : \mathit{OpenRange}, \mathit{overestimation}\ (\mathit{overlap}\ a\ b)$$

(defined in terms of things like $\mathit{overestimate}_{\leqslant_{CR}}$ shown in section 3.4) to over-estimate whether the following three intervals overlap:

1. $[0, \mathit{inf}\,]$
2. $\mathit{range\_inverse}\ \mathit{temp\_flow}\ r\_src\_temp\ r\_dst\_temp$
3. $\mathit{range\_inverse}\ \mathit{clock\_flow}\ r\_src\_clock\ r\_dst\_clock$

For a visual explanation, one may consult the left drawing in Figure 3 and view r_src_clock as $[a_1, b_1]$, r_dst_clock as $[c_1, d_1]$ etc. Overlap of 2 and 3 is equivalent to existence of a point in r_src from which one can flow to a point in r_dst. After all, if these two range inverses overlap, then there is a duration d that takes a certain temperature value in r_src_temp to a value in r_dst_temp and also takes a certain clock value in r_src_clock to a value in r_dst_clock. If 2 and 3 do *not* overlap, then either it takes so long for the temperature to flow from r_src_temp to r_dst_temp that any clock value in r_src_clock would "overshoot" r_dst_clock, or vice versa. Finally, if 1 does not overlap with 2 and 3, then apparently one could only flow backward in time, which is not permitted. Hence, overlap of these three ranges is a necessary condition for existence of concrete flow from points in r_src to points in r_dst, and so our $\mathit{abstract.cont\_trans}$ over-estimator may justifiably return "false" when the overlap over-estimator manages to prove absence of overlap.

4 Related Work

Verification of hybrid systems is an active field of research and there is a number of tools developed with this goal in mind; see [15] for a comprehensive list. Most of them are based on abstract refinement methods, either using box representations [20,17] or with polyhedra approximations [2,5,6].

Many of those tools are implemented in MATLAB [13] and those using some general programming language of choice most often rely on standard floating point arithmetic, which comes with its rounding errors. Some tools that address this problem include PHAVer [8], which relies on the Parma Polyhedra Library [3] for exact computations with non-convex polyhedra and HSolver [19], which is based on the constraint solver RSolver [18].

Formal verification becomes more and more important, especially in the field of hybrid systems, which are used to model safety critical systems of growing

complexity. There has been previous work on using general purpose theorem provers for verification of hybrid systems: see [1,10] and [12,4] for works using, respectively, PVS and STeP. KeYmaera [16] is a dedicated interactive theorem prover for specification and verification logic for hybrid systems. It combines deductive, real algebraic, and computer algebraic prover technologies and allows users to model hybrid systems, specify their properties and prove them in a semi-automated way.

However, to the best of our knowledge, none of the work or tools discussed above rely on a precise model of real number computations completely verified in a theorem prover, such as the model of CoRN used in this work.

5 Conclusions and Further Research

The presented verification of hybrid systems in Coq gives a nice showcase of proof-by-computation-on-computable-reals. The computable reals in CoRN do really complicated things for us, by approximating values for various real number expressions at great precision. The development also contains some nice layers of abstraction, involving the sophisticated use of type classes, e.g. the systematic use of estimators to make tactic-like optional-deciders, at each level in the stack and the use of the double negation monad.

It remains to be seen how far this automated verification approach can be taken, given the fact that we have limited ourselves to hybrid systems where we have a *solution* to the differential equation as a flow function, this flow function is *separable* (Section 2) and *range invertible* (Section 3.8) and the invariants are *stable* (Section 3.1). Finally, the reset functions should not be too strange, see [22] for details. If the flow functions are given, the largest part of the work is in producing *useful* range inverse functions. Note that we always have the trivial range inverse function, that just returns $\mathbf{R}$, but that is not useful. We want a function that actually helps to exclude certain abstract continuous transitions.

There is still a lot of room for more clever heuristics, possibly with less restrictive preconditions. The heuristic in [2] for bound selection doesn't work out of the box, but manual tweaking is obviously not ideal, so some more experimentation is required here. Finally, in case safety cannot be proved, one would like the system to generate an "offending trace" automatically, which can then be inspected by the user.

Acknowledgements. We thank the referees for their useful suggestions; due to space limitations we have not been able to incorporate all of them.

References

1. Ábrahám-Mumm, E., Hannemann, U., Steffen, M.: Assertion-based analysis of hybrid systems with PVS. In: Moreno-Díaz Jr., R., Buchberger, B., Freire, J.-L. (eds.) EUROCAST 2001. LNCS, vol. 2178, pp. 94–109. Springer, Heidelberg (2001)
2. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embedded Comput. Syst. 5, 152–199 (2006)

3. Bagnara, R., Hill, P., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. 72(1-2), 3–21 (2008)
4. Bjørner, N., Manna, Z., Sipma, H., Uribe, T.: Deductive verification of real-time systems using STeP. Theor. Comput. Sci. 253(1), 27–60 (2001)
5. Chutinan, A., Krogh, B.: Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 76–90. Springer, Heidelberg (1999)
6. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. 14(4), 583–604 (2003)
7. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-CoRN, the constructive Coq repository at Nijmegen. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 88–103. Springer, Heidelberg (2004)
8. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
9. Henzinger, T.: The theory of hybrid automata. pp. 278–292 (1996)
10. Henzinger, T., Rusu, V.: Reachability verification for hybrid automata. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386, pp. 190–204. Springer, Heidelberg (1998)
11. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Inf. Comput. 185(1), 105–157 (2003)
12. Manna, Z., Sipma, H.: Deductive verification of hybrid systems using STeP. In: Henzinger, T.A., Sastry, S.S. (eds.) HSCC 1998. LNCS, vol. 1386, pp. 305–318. Springer, Heidelberg (1998)
13. The MathWorks. MATLAB, http://www.mathworks.com/products/matlab/
14. O'Connor, R.: Certified exact transcendental real number computation in Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 246–261. Springer, Heidelberg (2008)
15. Pappas, G.: Hybrid system tools, http://wiki.grasp.upenn.edu/hst/
16. Platzer, A., Quesel, J.-D.: Keymaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
17. Preußig, J., Kowalewski, S., Wong-Toi, H., Henzinger, T.: An algorithm for the approximative analysis of rectangular automata. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 228–240. Springer, Heidelberg (1998)
18. Ratschan, S.: Efficient solving of quantified inequality constraints over the real numbers. ACM Transactions on Computational Logic 7(4), 723–748 (2006)
19. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. ACM Transactions in Embedded Computing Systems 6(1) (2007)
20. Stursberg, O., Kowalewski, S., Hoffmann, I., Preußig, J.: Comparing timed and hybrid automata as approximations of continuous systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S.S. (eds.) HS 1996. LNCS, vol. 1273, pp. 361–377. Springer, Heidelberg (1997)
21. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)
22. van der Weegen, E.: Automated machine-checked hybrid system safety proofs, an implementation of the abstraction method in Coq. Technical report, Radboud University Nijmegen (2009), http://www.eelis.net/research/hybrid/

Coverset Induction with Partiality and Subsorts: A Powerlist Case Study*

Joe Hendrix[1], Deepak Kapur[2], and José Meseguer[3]

[1] Galois, Inc.
[2] University of New Mexico
[3] University of Illinois at Urbana-Champaign

Abstract. Many inductive theorem provers generate induction schemes from the recursive calls appearing in terminating functions defined recursively in the specification. This strategy is called *coverset induction* in the context of algebraic specifications, and has been shown to be quite useful in a wide variety of applications. One challenge is that coverset induction typically requires using a total recursive function, while many operations are only meaningful on a subset of their inputs. A generalization of coverset induction method that supports partial constructors and operations specified in *membership equational logic* is proposed. The generalization has been implemented in the Maude ITP, and used to perform an extensive case study involving *powerlists* — a data structure introduced by J. Misra to elegantly formalize parallel algorithms based on divide and conquer strategy. Powerlists are constructed by partial operations, and it has been a challenge to naturally reason about powerlists using a formal logic that only supports total operations. We show how theorems about powerlists can be elegantly proven using the generalized coverset induction scheme implemented in the Maude ITP.

1 Introduction

Effective use of an inductive theorem prover requires efficient interaction between the user and the automated reasoning capabilities exposed by the tool. The user needs to be able to effectively direct how the tool uses its reasoning algorithms to enable the verification of systems that are intractable to fully automatic tools. Advanced reasoning algorithms are useful to minimize user interaction, but the user must also be able to debug failed proof attempts to identify lemmas that must be proven first or give additional hints to the theorem prover.

A key challenge in software verification is the treatment of partiality: many algorithms are designed to operate on data structures that satisfy specific invariants, while the mathematical logics underlying most work on automated theorem proving tools assume that the functions are total. When attempting to prove properties about the algorithms, the user must remember to carefully specify these invariant assumptions in a consistent way for all lemmas and correctness

* Research supported by NSF Grants CCF 0541315, CNS 0831462, and CNS 0905222.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 275–290, 2010.

properties; otherwise, the proof attempt can fail or inadvertently sidetrack the user into proving properties about special cases that were irrelevant to the context in which those procedures are used.

A logic that can support partiality in a natural and simple way may thus greatly ease the proving process. In this paper, we use *membership equational logic* (MEL) [5] for this purpose. In MEL, partiality is naturally represented by a distinction between *kinds* and *sorts*. Potentially meaningful terms have a kind; but only well-defined terms have a sort. Therefore, a term is *defined* iff it has a sort in the given specification. A term having a kind but lacking a sort is an *undefined* or *error* term. This provides a general logic for partial functions within a total (kinded) context, avoiding the complications of partial logics. Furthermore, sorts and subsorts can be hierarchically arranged in sort hierarchies; and the data elements of a sort can be defined by *semantic conditions* given by conditional memberships. Also, partial functions can become total on such semantically-defined sorts, so that typing of partial functions at the kind level can be further refined and overloaded by their restrictions to total functions at the sort level.

Since the Maude language [7] and its ITP inductive theorem prover are based on MEL, its logic is a good fit for reasoning inductively about functions and data structures involving partiality and subsorts. However, as we explain in this paper, a number of nontrivial improvements were needed to make it possible for the Maude ITP to support larger-scale case studies involving partial functions and partial data structures such as those appearing in the powerlist case study. The most crucial and theoretically interesting extension needed was perhaps the extension of *coverset induction* from its original total setting to the MEL setting supporting partiality and subsorts, and its integration into ITP. This contribution is the main focus of this paper. Another key extension is the mechanical support for the ability to reason about *alternative representations* of the same data structure (for example in arithmetic between numbers in Peano notation, $+$, or as products of primes). For lack of space, we do not explain other useful extensions such as better support for multiple equivalence relations and new commands to ease debugging, whose description can be found in [12].

We present some highlights of an extended case study involving powerlists, which is available from the ITP website [1]. Powerlists [21] are a data structure proposed by J. Misra to capture many data-parallel algorithms where lists of data may sometimes be appended, and sometimes interleaved to perform parallel operations in an algorithm. Their key feature is that such operations are partial, since powerlist operations require lists of the same length, which is always a power of 2. Mechanical reasoning about powerlists has been challenging, because of their partial nature. The main goal of our work is to make such inductive reasoning *as natural as possible*, so that mechanical proofs can match the simplicity of Misra's original hand proofs. In this regard our results are quite encouraging: we have proven many results about powerlists in a way that closely follow their hand proofs. The Maude ITP's support of partiality allowed us to stay focused on the heart of the proofs without being sidetracked on many

partiality and definedness issues. Although we needed some more lemmas than those explicitly proven in Misra's work to ensure that operations were well-defined, the proofs of such lemmas were mostly automatic.

The main contributions of the paper are: (i) a generalized form of coverset induction that naturally supports inductive reasoning with partiality, sorts, and subsorts; (ii) a new, publicly available version of the Maude ITP tool that implements these new inductive reasoning techniques; and (iii) an extensive case study on powerlists that validates the practical applicability of these methods to extensive and challenging reasoning problems.

In the next section, we review the relevant literature on coverset induction and reasoning about powerlists. In Section 2, we introduce membership equational logic (MEL), and show how powerlists can be defined in MEL. In Section 4, we present our generalization of coverset induction to MEL, and illustrate its use with a basic example from powerlists. In Section 5, we discuss implementation specific issues in the Maude ITP. In Section 6, we present an overview of our powerlist case study. Finally, we conclude in Section 7 with a summary and discussion of future work.

1.1 Related work

The cover set method for mechanizing inductive reasoning was proposed by Zhang et al [23] and served in the theorem prover *RRL* as the method for generating induction schemes from terminating and sufficient-complete recursive definitions specified as rewrite rules. It adapts NQThm's [6] approach of using the recursive calls to generate induction hypotheses. Other related methods for automatically generating induction schemes are based on the so-called *inductionless method* (also called Proof by Consistency by Kapur and Musser) [15, 22]. For checking consistency, test sets [3, 16] are generated from rewrite rules relating constructors of a data structure.

The implementation of the cover set method in RRL used a pattern of the form $f(x_1, \cdots, x_n)$, for generating an induction scheme from the definition of f. A generalization of the cover set method was proposed by Kapur and Subramaniam [18] for function definitions specified using $0, s, +$, where unification modulo linear arithmetic was used to generate induction schemes from patterns $f(s_1, \cdots, s_n)$, where the s_i's are linear arithmetic expressions. An orthogonal generalization was proposed by Kapur [14] for regular data structures specified using partial constructors with applicability conditions.

In proposing power lists [21], Misra presented well-crafted hand-proofs of properties of primitive operations of powerlists using algebraic laws, which were used in hand-proofs of some of the algorithms expressed using powerlists. This work was subsequently extended by Adams [2] for ripple-carry and carry-look-ahead adder circuits. Kapur [14] developed a method for specifying regular and semantic data structures by allowing constructors of such data structures to be partial using applicability conditions under which a constructor application is meaningful. This specification method served as the basis of reasoning about powerlists in preserving contexts as imposed by the applicability

conditions arising due the operations appearing in the rewrite rules. Properties of parametrized adder circuits of arbitrary data width were verified using RRL with minimal user guidance [19]. This work was subsequently extended to verify properties of parallel algorithms on hypercubes [17]. While the proofs generated by RRL closely mimics Misra and Adams' handcrafted proofs, they were done in a somewhat ad-hoc manner, primarily because, RRL does not support typing. A special inference method was developed to reason about the length properties of powerlists and contexts generated from the applicability conditions on top of a decision procedure for quantifier-free Presburger arithmetic implemented in the theorem prover. Consequently, for other semantic data structures with different applicability conditions, a user would have to start all over again from scratch, as RRL did not provide any typing support for specifying applicability conditions; further context-sensitive rewriting used for simplification using algebraic laws on regular and semantic data structures, as well as algorithms on these data structures had to be supported in an ad-hoc fashion.

Gamboa's formalization of powerlists in a quantifier-free first logic of recursive functions developed by Boyer and Moore is the closest in spirit to our formalization [10]. Gamboa specified powerlists as a subset of "dotted-pairs", much like the specification of the kind for powerlists here, and Misra's powerlists were a subset called *regular powerlists* and defined using similarity predicates. Operations were total and formalized in terms of destructor operations on powerlists. This formalization lead, in our opinion, to complex proofs that required many auxiliary lemmas to deal with ACL2's total logic. The theorems were perhaps more general, but clearly not as elegant as Misra's hand-coded proofs or proofs generated by RRL. However, Gamboa claimed that proofs generated by his approach have compositionality properties, critical in large mechanical verification. We believe that compositionality of proofs can be achieved while maintaining the simplicity and elegance of hand-proofs if order-sorted typing with membership equational logic is adopted as a basis for specifying regular and semantic data structures such as powerlists.

Bouhoula and Jacquemard have proposed using equational specifications made of rewrite rules with conditions and constraints where constraints are interpreted over constructor terms [4]. Tree grammars are used for automating proofs by induction about such equational specifications. Although we could not find any explicit proofs of properties of powerlists or algorithms on them, it is claimed there that this equational specification formalism can be used for regular and partial data structures such as powerlists and sorted lists.

2 Background about MEL Theories

We first review the underlying logic of the Maude ITP, called *Membership Equational Logic* (MEL), and show how powerlists can be formalized within it (see [5] for a more comprehensive introduction). Membership equational logic is essentially a many-kinded Horn logic with equality and unary predicates. The logic has two levels of typing: *kinds* type operator declarations in a signature, and *sorts* type terms using membership axioms in a particular theory.

Signatures. A *membership equational logic* (MEL) *signature* is a triple $\Sigma = (K, F, S)$ consisting of: (1) a set K of *kinds*; (2) a family of function symbols F typed by the kinds, where $f : k_1 \dots k_n \to k$ denotes that f is a function with n inputs of kind $k_1 \dots k_n$ and output kind k; and (3) a *disjoint* K-kinded family of sets $S = \{S_k\}_{k \in K}$ of *sorts*. Given a disjoint K-kinded family of *variables* $X = \{ X_k \}_{k \in K}$, where each set X_k is infinite, the *terms* of a MEL signature Σ are a K-indexed family $T_\Sigma(X) = \{ T_\Sigma(X)_k \}_{k \in K}$ where each set $T_\Sigma(X)_k$ denotes the set of well-kinded terms with kind k formed from the symbols in F and variables in X. We let $\mathrm{vars}(t)$ denote the variables appearing in a term $T_\Sigma(X)$, and let T_Σ denote the set of *ground terms* (terms without variables). A *substitution* is a mapping $\theta : Y \to T_\Sigma(X)$ with a finite domain $Y \subseteq X$ and the condition that $\theta(x_k) \in T_\Sigma(X)_k$ for each $x_k \in Y_k$.

Logic. Atomic formulas in MEL are either *equations* $t = u$ with $t, u \in T_\Sigma(X)_k$ for some $k \in K$ or *memberships* $t : s$ with $t \in T_\Sigma(X)_k$ and $s \in S_k$. A MEL theory $\mathcal{E}$ is a set of Horn clauses of the form $(\forall Y)\ \alpha$ **if** $\alpha_1 \wedge \dots \wedge \alpha_n$, where α and α_i $(i \in [1, n])$ are either equations or memberships with variables in Y. For a given atomic formula α, we write $\mathcal{E} \vdash \alpha$ if α can be derived via a sound and complete logic for simply-typed Horn clauses or MEL (e.g., [5]). Sorts in MEL are typically used to define a more refined notion of well-formed values in MEL than the kind-declarations. As syntactic sugar, we let $(\forall x_1 : s_1, \dots, x_n : s_n)\ \varphi$ denote the formula $(\forall x_1 \dots x_n)\, x_1 : s_1 \wedge \dots \wedge x_n : s_n \implies \varphi$. As noted in [5], each MEL theory $\mathcal{E}$ defines an initial algebra $T_\mathcal{E}$ whose elements are the equivalence classes of ground terms with respect to the relation $t =_\mathcal{E} u \iff \mathcal{E} \vdash t = u$. For any first-order formula φ over the atomic formulas in Σ, we write $T_\mathcal{E} \models \varphi$ if φ is true in $T_\mathcal{E}$. Inductive reasoning is reasoning in $T_\mathcal{E}$.

MEL specifications can be executed by orienting equations $l = r$ left-to-right as rules, and using term rewriting techniques extended to support the membership constraints as well (see [5] for details). In general, rewriting can lead to non-termination and non-confluence. In this paper, we assume that the Maude specification is both operationally terminating [9] and confluent [5]. These properties guarantees agreement between the equational semantics, and the operational semantics obtained by rewriting. We rely on existing tools available for checking termination and confluence [8]. Operational termination also guarantees the soundness of the coverset induction.

3 Specification of Powerlists as a MEL Theory

We formalize unnested powerlists [21] over natural numbers as a MEL theory Powerlists are non-empty lists formed from singleton lists $[i]$ and two binary operations: an operation $\mid$ called *tie* that concatenates its arguments, and a second operation $\times$ called *zip* that interleaves its arguments together. Both operations require that their arguments are *similar* (i.e.,to have the same length), and consequently the length is always a power of 2. For example, both $(1 \mid 2) \mid (3 \mid 4)$ and $(1 \mid 2) \times (3 \mid 4) = (1 \mid 3) \mid (2 \mid 4)$ are well-formed powerlists, while $(4 \mid 3) \times 1$ and $(1 \mid 2) \mid 3$ are not well-formed.

```
fmod POWERLIST is protecting NAT .
  sort Pow .
  op [_] : Nat -> Pow [ctor].
  op _|_ : [Pow] [Pow] -> [Pow] .

  vars M N : Nat .    vars P Q R S : Pow .

  cmb P | Q : Pow if sim?(P, Q) = true .

  op sim? : Pow Pow -> Bool [comm].
  eq sim?(P | Q, R | S) = sim?(P, R) .
  eq sim?([M], [N]) = true .
  eq sim?(P | Q, [N]) = false .

  op _x_ : [Pow] [Pow] -> [Pow] .
  eq (P | Q) x (R | S) = (P x R) | (Q x S) .
  eq [M] x [N] = [M] | [N] .
endfm
```

Fig. 1. Powerlist example in Maude syntax

We formalize unnested powerlists in the `POWERLIST` module defined in Figure 1. The module `POWERLIST` imports the predefined module `NAT` that defines a sort `Nat` for representing the natural numbers, along with common operations on them. We introduce the sort `Pow`, which we will reserve for those terms representing powerlists; Maude automatically introduces also the kind `[Pow]` to denote the kind of the sort `Pow`. We also introduce four operators: `[_]` for representing the operation that forms powerlist elements; `_|_` for representing the powerlist *tie* operation; `_x_` for representing the powerlist *zip* operation; and `sim?` for representing the similarity relation that holds on two well-sorted powerlists if they have the same length. Similarity is declared to be commutative to best take advantage of Maude's support for rewriting modulo axioms, and uses the sort constraints on variables to simplify its definition.

In the variable declaration section, we associate the sort `Nat` to the variables `M` and `N`, and the sort `Pow` to the variables `P`, `Q`, `R`, and `S`. By doing this, we are in fact declaring: (i) that `M` and `N` are variables of the kind `[Nat]`, and `P`, `Q`, `R`, and `S` of the kind `[Pow]`, and (ii) that in all memberships and equations in which those variables appear, there is an implicit extra condition stating that those variables only range over the set of terms belonging to their associated sort. Finally, in the membership declaration section, we declare that the *tie* of two powerlists are powerlists if they have equal length. In fact, the declarations in the module:

```
op [_] : Nat -> Pow [ctor] .
op sim? : Pow Pow -> Bool [comm].

eq sim?(P | Q, R | S) = sim?(P, R) .
```

are just syntactic sugar for the following:

```
op [_] : [Nat] -> [Pow] .
cmb [N] : Pow if N : Nat.
op sim? : [Pow] [Pow] -> [Bool] [comm].
cmb sim?(P, Q) : Bool if P : Pow /\ Q : Pow .

ceq sim?(P | Q, R | S) = sim?(P, R)
 if P : Pow /\ Q : Pow /\ R : Pow /\ S : Pow .
```

Since not all terms constructed with the operators `_|_` and `_x_` will represent powerlists, those operators are declared at the kind level. We have a membership for the *tie* operation, but can omit one for the *zip* operation, because zip is defined equationally in terms of tie. The membership for zip is thus an inductive consequence of the other axioms, and is straightfoward to prove in the Maude ITP using the coverset induction techniques described in the next section.

Given arbitrary terms t and u with kind `[Pow]`, $t \mid u$ denotes a term with kind `[Pow]`. However, $t \mid u$ only has sort `Pow` if it matches a membership with sort `Pow`. In the `POWERLIST` module, this requires that t and u both have sort `Pow` and $\texttt{sim?}(t, u) = \texttt{true}$. As an example, the term `[1] | ([2] | [3])` is not a powerlist. This is represented in `POWERLIST` by the fact that the term `([1] | [2]) | [3]` has kind `[Pow]`, but it does not belong to the sort `Pow` as `sim?([1], [2] | [3]) = false`. When given terms that are not proper powerlists, `sim?` may return `true` even if the terms have different lengths. However, such terms are not well-sorted, and thus are irrelevant in proofs involving well-sorted powerlists. For example, even though `sim?([1] | [2], [3] | ([4] | [5])) = true`, the conditional membership defining the sort `Pow` *cannot* be applied to the term `([1] | [2]) | ([3] | ([4] | [5]))`, because the subterm `[3] | ([4] | [5])` does not have sort `Pow`.

In general, sort declarations on defined operations such as `_x_` or `sim?` should be viewed as *assertions* rather than axioms, and they should be an inductive consequence of the other axioms. One can either prove such assertions with the Maude ITP, or in some cases, completely automatically with the Maude Sufficient Completeness Checker [13]. By default, memberships implied by operator declarations are viewed as assertions, however they are treated as axioms if one uses the `ctor` attribute (as done in the case with `[_]` above).

4 Generalizing Coverset Induction in MEL Theories

In order to perform proofs by induction, we need a well-founded ordering that can be used for an induction scheme and for generating induction hypotheses. Coverset induction [23] generates induction schemes for recursive operations defined by a complete set of terminating rewrite rules. The theorem prover typically generates a separate goal for each rewrite rule used to define an operation: equations with no recursive calls generate base cases, while recursive equations yield induction steps.

Traditional coverset induction works quite well for total operations. However, it is unsound in general for partial operations, because no case is generated when the operation is undefined. For partial operations, a sound algorithm for performing coverset induction must either generate additional base cases to cover

arguments where the operation is undefined, or requires a separate completeness property which implies the undefined cases are irrelevant under certain assumptions about the current theory. We follow the first approach in this paper, because it better supports a more general notion of partiality where the theory has complete freedom to define the meaning of sorts axiomatically. The second approach has been followed in [14], where the partiality constraints where expressed as part of the signature.

Assumptions. Our coverset induction algorithm works by incrementally instantiating variables in a pattern term to match the left-hand sides of equations in a MEL theory $\mathcal{E}$ that is operationally terminating and normalizing when oriented as a rewrite theory [5, 9]. For our coverset algorithm to terminate, we require that the only memberships over variables $x : s$ **if** $x : s'$ are those used to define a partial order $<$, called the *subsort ordering*, over the sorts in the theory. This restriction is satisfied in practice by Maude specifications.

We also assume that every variable appearing in a conditional membership or equation is constrained to have a specific sort. If needed, one could transform a specification into an equivalent specification satisfying this property by (1) introducing a fresh maximal sort s_k for each kind k, (2) adding memberships so that each term with kind k has sort s_k, and (3) for each formula φ **if** $\overline{\alpha} \in \mathcal{E}$, adding extra constraint $x : s_k$ to $\overline{\alpha}$ when there is no sort constraint $x : s \in \overline{\alpha}$ and k is the kind of variable x.

In our algorithm, we assume that the user is attempting to prove inductively that $T_{\mathcal{E}} \models (\forall x_1 : s_1 \ldots x_n : s_n)\, \varphi$. To help motivate this with a running example, suppose that we have added a function for returning the last element in a powerlist to the specification `POWERLIST` of powerlists given in Figure 1:

```
op last : Pow -> Nat .
eq last(P | Q) = last(Q) .  eq last([N]) = N .
```

We want to prove that last could be defined using the zip operator as well:

```
A{P:Pow ; Q:Pow} ((sim?(P, Q)) = (true) => (last(P x Q)) = (last(Q)))
```

This theorem is given in Maude ITP syntax where the quantified variables `P` and `Q` appear with their sorts defined in the universally quantified formula. The extra parenthesis are needed for parsing Maude terms within the formula. In desugared mathematical syntax, the above formula is equivalent to

$$(\forall p, q)\ p : \mathtt{Pow} \wedge q : \mathtt{Pow} \wedge \mathtt{sim?}(p, q) = \mathtt{true} \implies \mathtt{last}(p\, \mathtt{x}\, q) = \mathtt{last}(q)$$

The first step for generating a coverset induction scheme is to identify a term $p \in T_\Sigma(X)$ called the *pattern*, with free variables $Y = \{ x_1, \ldots, x_n \}$. It is not required, but the pattern is typically a subterm in φ whose root symbol is a recursively defined function. The equations whose left-hand sides match the pattern are used to generate induction cases. For our running example, we use the pattern `P x Q`. The relevant equations are

```
eq (P | Q) x (R | S) = (P x R) | (Q x S) .
eq [M] x [N] = [M] | [N] .
```

Our algorithm for generating coverset induction schemes performs constructor-based narrowing [11] to match the initial pattern p against the left-hand sides of equations appearing in the specification. The result is a set of induction cases $\{ \varphi_1, \ldots \varphi_m \}$ such that

$$T_{\mathcal{E}} \models (\forall x_1 : s_1 \ldots x_n : s_n)\, \varphi \iff T_{\mathcal{E}} \models \varphi_1 \wedge \cdots \wedge \varphi_m. \tag{1}$$

By itself, the property in (1) is trivial to satisfy — the set $\{ \varphi \}$ would satisfy this property. The usefulness of coverset induction stems from the fact that coverset induction should generate formulas $\varphi_1, \ldots, \varphi_m$ that have terms that are *reducible* from the equations in the specification, and have induction hypotheses that are applicable to the resulting normalized subterms.

Our coverset induction algorithm consists of two phases: a *goal narrowing* phase, and a *case generation* phase. The goal narrowing phase analyzes the left-hand sides of the equations in $\mathcal{E}$ to generate a set of *substitutions* that are used to generate the induction cases. The case generation phase further analyzes the right-hand sides of recursive equations to find appropriate induction hypotheses for each case.

Goal narrowing. The induction cases are generated by instantiating the formula φ in various ways. We define how to generate the initial substitutions used to generate the goals. A *conditional term* is a pair (t, α) where $t \in T_\Sigma(X)$ and α is a conjunction of atomic formulas that constrains the variables appearing in the right-hand side of $\mathrm{vars}(t)$. We say that a ground term $u \in T_\Sigma$ is a *ground instance* of (t, α) iff there is a substitution $\theta : X \to T_\Sigma$ such that: (1) $u = t\theta$ and (2) $T_{\mathcal{E}} \models \alpha\theta$. Given a disjunctive set Δ of conditional terms, we say that u is a ground instance of Δ iff it is an instance of some conditional term $(t, \alpha) \in \Delta$.

Our narrowing procedure is an iterative procedure that repeatedly applies an inference rule defined below on a disjunctive set of conditional substitutions. We start with the initial set $\Delta_p^0 = \{ (p, x_1 : s_1 \wedge \cdots \wedge x_n : s_n) \}$, and apply the rule (2) defined below until termination to obtain $\Delta_p^1, \Delta_p^2, \ldots, \Delta_p^*$. In our running example, we have $\Delta_{\texttt{P x Q}}^0 = \{ (\texttt{P x Q},\ \texttt{P} : \texttt{Pow} \wedge \texttt{Q} : \texttt{Pow}) \}$.

From a semantic perspective, the ground instances ψ of Δ_p^0 correspond to the possible ground instantiations of the pattern p using the sort constraints in the initial formula φ. Our approach is to incrementally replace variables in conditional terms with constructor instances to match the left-hand-sides of rewrite rules in $\mathcal{E}$. To find variables to instantiate, we introduce the concept of the *demanded variables* of a term — that is, variables in a term whose instantiation may cause equations to match the term. Formally, the demanded variables of a $t \in T_\Sigma(X)$ is the set $\mathrm{dvars}(t) \subseteq \mathrm{vars}(t)$ such that

$$x \in \mathrm{dvars}(t) \iff \exists\, (l = r \textbf{ if } \alpha) \in \mathcal{E} \text{ s.t. } \mathrm{mgu}(t, l) = \theta \wedge \theta(x) \notin X.$$

where $\mathrm{mgu}(t, l)$ returns the most general unifier of t and l if it exists and $\bot$ otherwise. For the initial pattern `P x Q` in the example, $\mathrm{dvars}(\texttt{P x Q}) = \{ \texttt{P}, \texttt{Q} \}$.

We can then define our inference rule as follows:

$$\frac{(t, \overline{\alpha}) \in \Delta_p^i \quad x : s \in \overline{\alpha} \quad x \in \mathrm{dvars}(t)}{\Delta_p^{i+1} := \Delta_p^i \setminus \{ (t, \overline{\alpha}) \} \cup \{ (t[x/u], \overline{\alpha}[x/u] \wedge \overline{\alpha}') \mid u : s \textbf{ if } \overline{\alpha}' \in \mathcal{E} \}} \tag{2}$$

The final set Δ_p^* is then used to generate induction cases. This process is guaranteed to terminate, but is not confluent. Different values for Δ_p^* may be obtained depending on which variable is expanded in each application. We will discuss the strategy chosen by the Maude ITP when we describe the implementation in the next section.

In our example, we can expand either `P` or `Q` in the initial set $\Delta^0_{\mathtt{PxQ}}$. In this case we choose to expand p. The relevant memberships are

```
cmb [N] : Pow if N : Nat.
cmb P | Q : Pow if sim?(P, Q) = true /\ P : Pow /\ Q : Pow .
```

Expanding using these memberships results in the set

$$\Delta^1_{\mathtt{PxQ}} = \{\ ([\mathtt{M}] \mid \mathtt{Q},\quad \mathtt{M} : \mathtt{Nat} \wedge \mathtt{Q} : \mathtt{Pow}),\\ ((\mathtt{P}_1 \mid \mathtt{P}_2)\ \mathtt{x}\ \mathtt{Q},\quad \mathtt{P}_1 : \mathtt{Pow} \wedge \mathtt{P}_2 : \mathtt{Pow} \wedge \mathtt{Q} : \mathtt{Pow} \wedge \mathtt{sim?}(\mathtt{P}_1, \mathtt{P}_2) = \mathtt{true})\ \}$$

The variable Q is expandable in both of the substitutions in $\Delta^1_{\mathtt{PxQ}}$ due to the first and second equations on $\times$ respectively. Expanding each substituion in order yields the sets

$$\begin{aligned}
\Delta^2_{\mathtt{PxQ}} = \{\ & ([\mathtt{M}]\ \mathtt{x}\ [\mathtt{N}],\quad \mathtt{M} : \mathtt{Nat} \wedge \mathtt{N} : \mathtt{Nat}),\\
& ([\mathtt{M}]\ \mathtt{x}\ (\mathtt{Q}_1 \mid \mathtt{Q}_2),\quad \mathtt{M} : \mathtt{Nat} \wedge \mathtt{Q}_1 : \mathtt{Pow} \wedge \mathtt{Q}_2 : \mathtt{Pow} \wedge \mathtt{sim?}(\mathtt{Q}_1, \mathtt{Q}_2) = \mathtt{true}),\\
& ((\mathtt{P}_1 \mid \mathtt{P}_2)\ \mathtt{x}\ \mathtt{Q},\quad \mathtt{P}_1 : \mathtt{Pow} \wedge \mathtt{P}_2 : \mathtt{Pow} \wedge \mathtt{Q} : \mathtt{Pow} \wedge \mathtt{sim?}(\mathtt{P}_1, \mathtt{P}_2) = \mathtt{true})\ \}\\
\Delta^3_{\mathtt{PxQ}} = \{\ & ([\mathtt{M}]\ \mathtt{x}\ [\mathtt{N}],\quad \mathtt{M} : \mathtt{Nat} \wedge \mathtt{N} : \mathtt{Nat}),\\
& ([\mathtt{M}]\ \mathtt{x}\ (\mathtt{Q}_1 \mid \mathtt{Q}_2),\quad \mathtt{M} : \mathtt{Nat} \wedge \mathtt{Q}_1 : \mathtt{Pow} \wedge \mathtt{Q}_2 : \mathtt{Pow} \wedge \mathtt{sim?}(\mathtt{Q}_1, \mathtt{Q}_2) = \mathtt{true}),\\
& ((\mathtt{P}_1 \mid \mathtt{P}_2)\ \mathtt{x}\ [\mathtt{N}],\quad \mathtt{P}_1 : \mathtt{Pow} \wedge \mathtt{P}_2 : \mathtt{Pow} \wedge \mathtt{N} : \mathtt{Nat} \wedge \mathtt{sim?}(\mathtt{P}_1, \mathtt{P}_2) = \mathtt{true}),\\
& ((\mathtt{P}_1 \mid \mathtt{P}_2)\ \mathtt{x}\ (\mathtt{Q}_1 \mid \mathtt{Q}_2),\quad \mathtt{P}_1 : \mathtt{Pow} \wedge \mathtt{P}_2 : \mathtt{Pow} \wedge \mathtt{Q}_1 : \mathtt{Pow} \wedge \mathtt{Q}_2 : \mathtt{Pow}\\
& \quad \wedge \mathtt{sim?}(\mathtt{P}_1, \mathtt{P}_2) = \mathtt{true} \wedge \mathtt{sim?}(\mathtt{Q}_1, \mathtt{Q}_2) = \mathtt{true})\ \}
\end{aligned}$$

The algorithm terminates with $\Delta^3_{\mathtt{PxQ}}$, because the variables are appearing in terms are not further expandable.

It is not difficult to show that each application of the inference rule preserves the ground instances of Δ_p^i.

Proposition 1. *If Δ_p^{i+1} is obtained from Δ_p^i by applying the inference rule (2), then Δ_p^{i+1} has the same set of ground instances as Δ_p^i.*

Case generation. Finally, we generate an induction case $\varphi_{t,\alpha}$ from each conditional substitution in Δ_p^*. The induction hypotheses are obtained by matching t against the left-hand sides of equations in $\mathcal{E}$. Given terms $t, u \in T_\Sigma(X)$, we let $match(t, u)$ denote a substitution θ s.t. $t\theta = u$ if one exists, and $\perp$ otherwise. It is not difficult to see that each conditional substitution $(t, \overline{\alpha}) \in \Delta_p^*$, matches p, and we let $\theta = \text{match}(p, t)$ The formula $\varphi_{t,\alpha}$ is defined as follows:

$$\varphi_{t,\overline{\alpha}} \quad = \quad \Big(\overline{\alpha} \wedge \bigwedge_{\substack{l=r\ \textbf{if}\ \overline{\alpha}' \in \mathcal{E}\\ \psi = \text{match}(l,t)}} \ \bigwedge_{\substack{u \in \text{subterms}(r\psi)\\ \rho = \text{match}(p,u)}} (\overline{\alpha}'\psi \implies \varphi\rho)\Big) \implies \varphi\theta$$

Coverset induction on our example `P x Q` yields the 4 subgoals which can be discharged automatically within the Maude ITP via rewriting.

```
A{M:Nat ; N:Nat}
 ((sim?([M], [N])) = (true) => (last([M] x [N])) = (last([N])))

A{M:Nat ; Q1:Pow ; Q2:Pow}
 ((sim?([M], Q1 | Q2)) = (true)
  => (last([M] x (Q1 | Q2))) = (last(Q1 | Q2)))

A{P1:Pow ; P2:Pow ; N:Nat}
 ((sim?(P1 | P2, [N])) = (true)
  => (last((P1 | P2) x [N])) = (last([N])))

A{P1:Pow ; P2:Pow ; Q1:Pow ; Q2:Pow}
 ((sim?(P1 | P2, Q1 | Q2)) = (true)
  & ((sim?(P1, Q1)) = (true) => (last(P1 x Q1)) = (last(Q1)))
  & ((sim?(P2, Q2)) = (true) => (last(P2 x Q2)) = (last(Q2)))
  => (last((P1 | P2) x (Q1 | Q2))) = (last(Q1 | Q2)))
```

We show the correctness of our coverset induction algorithm in the following:

Theorem 1 ([12]). *Let $\mathcal{E}$ denote a MEL theory that is operationally terminating, ground confluent and ground sort-decreasing. For each formula with the form $(\forall x_1 : s_1 \ldots x_n : s_n)\ \varphi$ and a pattern $p \in T_\Sigma(X)$ with variables $\{x_1, \ldots, x_n\}$, let Δ_p^* be a set of conditional substitutions obtained by applying the inference rule (2) until completion starting from Δ_p^0.*

$$T_\mathcal{E} \models \varphi \iff \bigwedge_{(t,\overline{\alpha}) \in \Delta_p^*} T_\mathcal{E} \models \varphi_{t,\overline{\alpha}}.$$

In the initial coverset induction approach of the RRL, the initial pattern p was required to have the form $f(x_1, \ldots, x_n)$ where f was a total defined function. Under these restrictions, the coverset induction schemes can be generated directly form the equations used to define f without the goal narrowing phase presented here. Our separate goal narrowing phase allows both support for partial functions and more general patterns.

5 Implementation in the Maude ITP

The Maude ITP tool is an interactive theorem prover capable of showing that a first-order formula holds in the initial model $T_\mathcal{E}$ of an MEL theory $\mathcal{E}$ given as a Maude specification. The ITP is written in Maude, and uses Maude's extensive support for *reflection* to reason about Maude theories. This support for reflection is critical to other Maude-based reasoning tools such as the Maude Termination Tool and confluence checker [8].

We have extended the Maude ITP with two commands `cov` and `cov*` which apply coverset induction when invoked by the user. The difference between the two commands is that `cov*` will automatically simplify all of the subgoals generated by coverset induction using Maude's built-in tactics, while `cov` will leave them unchanged. The commands have the syntax:

`(cov on` *pattern*`)` `(cov* on` *pattern*`)`

While using coverset induction in the powerlist case study, we found several ways to improve the implementation over a naive implementation suggested by the theoretical work in the previous section: (1) allow the user to define *alternative constructors* for sorts in the specification; (2) allow substitutions in subgoals to be further specialized by *additional patterns*; and (1) apply the Δ-rule only to variables that are *demanded* by the largest number of equations in $\mathcal{E}$. We also perform a simple syntactic *subsumption* test to eliminate redundant induction cases.

Alternative constructors. Many proofs in mathematics and computer science rely on the ability to view the same data in multiple ways. This includes the natural numbers, which can be viewed in terms of zero and successor, or the product of primes. It also applies to the powerlist data structure which we will describe in more detail in the next section. To accommodate this in the ITP, we added a command `ctor-def` which allows one to define an alternative named set of memberships $t_1 : s$ **if** $\overline{\alpha}_1 \ldots t_n : s$ **if** $\overline{\alpha}_n$ whose general form introduces a proof obligation of the following form:

$$(\forall x : s) \quad (\exists \overline{y}_1 . \, t_1 = x \wedge \overline{\alpha}_1) \vee \cdots \vee (\exists \overline{y}_n . \, t_n = x \wedge \overline{\alpha}_n)$$

By proving this conjecture, one can optionally have the ITP use the alternative memberships in later proofs.

The ability to use alternative constructors is crucial to our powerlist case study. In the `POWERLIST` module in Section 2, one could define alternative constructors for sort `Pow` using zip:

```
cmb [M] : Pow if M : Nat .
cmb (P x Q) : Pow if  P : Pow /\ Q : Pow /\ sim?(P, Q) = true .
```

As a simple example, we introduce operations for rotating elements in a powerlist to the right and left:

```
op rr : Pow -> Pow .          op rl : Pow -> Pow .
eq rr([N]) = [N] .            eq rl([N]) = [N] .
eq rr(P x Q) = rr(Q) x P .    eq rl(P x Q) = Q x rl(P) .
```

After proving an alternative constructors for zip and a few lemmas to ensure the term `rr(P x Q)` is well-sorted, the proof that $\mathrm{rl}(\mathrm{rr}(p)) = p$ goes through with a single coverset induction command: `(cov* using zip on rr(P) .)`

Additional patterns. One useful feature of coverset induction is that, in addition to generating potentially useful induction hypotheses, it instantiates terms appearing in the current problem to match additional rules in the specification. This allows simplification by rewriting, and may be necessary for the ITP to automatically resolve goals. To aid in this, the Maude ITP allows the user to specify additional pattern terms that further refine substitutions generated by the *case generation* phase.

Most-demanded variable heuristic. As mentioned previously, our coverset induction rule is not confluent, and may yield different induction schemes

depending on which variables are instantiated. As a heuristic, if multiple variables in a conditional substitution are eligible for expansion by the rule (2), we expand the variable demanded by a maximal number of equations.

As an example, we define a function contains which returns true if a given natural number appears in a powerlist.

```
op count : Nat Pow -> Bool .
eq count(N, P | Q) = count(N, P) or count(N, Q) .
eq count(0, [0])   = true .   eq count(s N, [s M]) = count(N, [M]) .
eq count(0, [s M]) = false .  eq count(s N, [0])   = false .
```

In performing coverset induction using the pattern `count(N, P)`, the Maude ITP expands `P` before expanding `N`. `P` is demanded by all 5 equations while `N` is demanded by only 4. This optimization eliminates an extra case, and simplified the verification of adder circuits done as part of our powerlist case study.

6 Powerlist Case Study

In this section, we briefly discuss how the work in the previous two sections has been applied to the verification of algorithms over powerlists [21]. We have used the ITP to mechanically prove a wide variety of theorems dealing with powerlists — including Misra's theorems in [21] on basic properties of powerlists, Fast Fourier Transform, Batcher's sorting, parallel prefix-sum, as well as results in [2] on the correctness of arbitrary-width ripple-carry and carry-lookahead adders. Overall, our case study has lead us to two major conclusions: (1) the ITP would definitely benefit from implementing some of the proof management capabilities, decision procedures, and support for parameterized theories available in other theorem provers; (2) despite these limitations, the use of membership equational logic helped lead to relatively simple and straightforward proofs.

A more comprehensive look at our case study is available in [12]. Due to space constraints, we omit a detailed look of each proof here. Instead, we will focus on one example — the correctness of Ladner and Fischer's parallel prefix sum algorithm [20]. Misra showed in [21] how it could easily be formalized using powerlists. Given a powerlist p over scalars and a binary operation $+$, the prefix sum $\mathrm{ps}(p)$ is defined by:

$$\mathrm{ps}([x_0, x_1, \ldots, x_n]) = [x_0, x_0 + x_1, \ldots, x_0 + x_1 + \cdots + x_n]$$

A sequential definition of prefix sum can be defined in Maude as follows:

```
op ps : Pow -> Pow .
eq ps([ N ]) = [ N ] .
eq ps(P | Q) = ps(P) | (last(ps(P)) + ps(Q)) .
```

where $\mathrm{last}(P)$ returns the last element of P (see Section 4), and $+$ has been overloaded to extend it to powerlists:

```
op _+_ : Nat Pow -> Pow .
eq M + (P | Q) = (M + P) | (M + Q) .
eq M + [ N ] = [ M + N ] .
```

Operationally, our definition of `ps` computes the prefix sum of the left half of the powerlist, then adds the total to the prefix sum of the right half.

When the operation + is associative and has identity 0, Ladner and Fischer [20] showed that the prefix sum can be efficiently computed in parallel by computing the sums of adjacent elements recursively. Given a powerlist $p \times q$, we let $p + q$ denote the elementwise sum of p and q. To compute $\mathrm{ps}(p \times q)$, the Ladner and Fischer algorithm computes the sum $\mathrm{ps}(p + q)$ which has half as many elements, and then uses this sum to compute the final result in parallel. To define this in Maude, we first introduce an operation for inserting an element into a powerlist, and shifting the remaining elements to the right:

```
op rsh : Nat Pow -> Pow .
eq rsh(M, [ N ]) = [ M ] .
eq rsh(M, P | Q) = rsh(M, P) | rsh(last(P), Q) .
```

We can then express the Ladner-Fischer closure as follows:

```
op lf : Pow -> Pow .
eq lf([ N ]) = [ N ] .
eq lf(P x Q) = (rsh(0, lf(P ++ Q)) + P) x lf(P ++ Q) .
```

where ++ is the elementwise powerlist sum:

```
op _++_ : [Pow] [Pow] -> [Pow] .
eq (P1 | P2) ++ (Q1 | Q2) = (P1 ++ Q1) | (P2 ++ Q2) .
eq [ M ] ++ [ N ] = [ M + N ] .
```

Our ITP proof that `ps` and `lf` are equivalent is quite straightforward, and starts with the following command:

```
(goal ps-lf : POWERLIST-PREFIX |- A{P:Pow} ((ps(P)) = (lf(P))) .)
```

To reduce space, we will assume that we have already shown some basic theorems about powerlist similarity, the zip operation, the last operation and basic types (a self-contained proof script is available at [1]). We next introduce an associativity lemma and a lemma to commute `rsh` and +:

```
(lem elt-sum-assoc :
  A{N:Nat ; P:Pow ; Q:Pow}
   ((sim?(P,Q)) = (true) => ((N + P) ++ Q) = (N + (P ++ Q))) .)
(cov* on P ++ Q .)

(lem rsh-self-elt-plus :
  A{N:Nat ; P:Pow} ((rsh(N, N + P)) = (N + rsh(0, P))) .)
  (lem rsh-self-elt-plus-general :
    A{M:Nat ; N:Nat ; P:Pow}
     ((rsh(N + M, N + P)) = (N + rsh(M, P))) .)
  (cov* on N + P .)
(cov* on N + P .)
```

The previous lemmas simplify subterms appearing in the right-hand side of the definition of `ps`. By using these lemmas, we can easily show that `ps` can be defined in terms of zip:

```
(lem ps-zip :
  A{P:Pow ; Q:Pow}
   ((sim?(P,Q)) = (true) =>
    (ps(P x Q)) = ((rsh(0, ps(P ++ Q)) + P) x ps(P ++ Q))) .)
(cov* on P ++ Q .)
```

The main goal is then solved with: `(cov* using zip on lf(P) .)`

In Misra's original proof [21], the correctness of Ladner and Fischer's algorithm was approached by defining a *simple prefix sum*, which used the zip operation instead of tie. He then showed how Ladner and Fischer's scheme could be derived with a few lemmas and a small number of algebraic steps. This derivation involved defining a recursive powerlist equation which was then solved for the correct solution. It would be interesting to see if this approach can be more fully automated as the Maude ITP is only able to verify properties about existing algorithms, rather than deriving an algorithm by solving equations. We suspect that such work would require a specialized constraint solver, pehaps a unification procedure, that has been tailored to directly supported the powerlist operations. In Ruben Gamboa's work with ACL2 [10], he also introduced simple prefix sums, but his proof in ACL2 required a number of auxiliary lemmas to deal with ACL2's inability to treat zip a constructor.

7 Conclusions

Our work has addressed the need to reason inductively about partial data structures and partial operations. Based on MEL's natural and simple support for subsorts and partiality, we have presented a generalization of the coverset induction method that naturally supports reasoning on such partial operations without cluttering the user with a host of definedness proofs alien to the heart of the reasoning. We have implemented these methods in a new version of the Maude ITP tool and have demonstrated the simplicity and naturalness of this approach by means of an extensive powerlist case study. Due to space limitations, we could only give a small sampling of the theorems proved. Many more theorems were proven in [12], with their code and tool publicly available in [1].

For future work, greater support for inductive reasoning about parametrized modules in the Maude ITP would have been very helpful and should be added to a new version of the tool. Also, simpler proofs could clearly be fully automated. Perhaps more importantly, we hope that this work will stimulate us and others to study how support for types, subtypes and partiality can be added to inductive theorem provers already supporting typed logics, so that such reasoning becomes as simple and natural as possible.

Acknowledgments. The authors would like to thank the anonymous referees for suggestions that helped improve the quality of this paper.

References

[1] Maude ITP website at UIUC, `http://maude.cs.uiuc.edu/tools/itp/`

[2] Adams, W.: Verifying adder circuits using powerlists. Technical report, University of Texas, Austin, Department of Computer Science, Austin, TX, USA (1994)

[3] Bouhoula, A.: Automated theorem proving by test set induction. J. Symb. Comput. 23(1), 47–77 (1997)
[4] Bouhoula, A., Jacquemard, F.: Automatic verification of sufficient completeness for conditional constrained term rewriting systems. Technical Report LSC-05-17, ENS de Cachan (2006), http://www.lsv.ens-cachan.fr/Publis/
[5] Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Comput. Sci. 236, 35–132 (2000)
[6] Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, London (1979)
[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
[8] Clavel, M., Durán, F., Hendrix, J., Lucas, S., Meseguer, J., Ölveczky, P.C.: The Maude formal tool environment. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 173–178. Springer, Heidelberg (2007)
[9] Durán, F., Lucas, S., Marché, C., Meseguer, J., Urbain, X.: Proving operational termination of membership equational programs. Higher-Order and Symbolic Computation 21(1-2), 59–88 (2008)
[10] Gamboa, R.: Defthms about zip and tie: Reasoning about powerlists in ACL2. Technical Report TR87-02, University of Texas Computer Science (1997)
[11] Hanus, M.: The integration of functions into logic programming: From theory to practice. J. Log. Program. 19/20, 583–628 (1994)
[12] Hendrix, J.: Decision Procedures for Equationally Based Reasoning. PhD thesis, University of Illinois at Urbana-Champaign (2008)
[13] Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 151–155. Springer, Heidelberg (2006)
[14] Kapur, D.: Constructors can be partial too. In: Automated Reasoning and its Applications: Essays in Honor of Larry Wos, pp. 177–210. MIT Press, Cambridge (1997)
[15] Kapur, D., Musser, D.R.: Proof by consistency. Artif. Intell. 31(2), 125–157 (1987)
[16] Kapur, D., Narendran, P., Zhang, H.: Automating inductionless induction using test sets. Journal of Symbolic Computation 11, 83–111 (1991)
[17] Kapur, D., Subramaniam, M.: Automated reasoning about parallel algorithms using powerlists. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 416–430. Springer, Heidelberg (1995)
[18] Kapur, D., Subramaniam, M.: New uses of linear arithmetic in automated theorem proving by induction. J. Automated Reasoning 16(1-2), 39–78 (1996)
[19] Kapur, D., Subramaniam, M.: Mechanical verification of adder circuits using Rewrite Rule Laboratory. Formal Methods in System Design 13(2), 127–158 (1998)
[20] Ladner, R.E., Fischer, M.J.: Parallel prefix computation. Journal of the ACM 27(4), 831–838 (1980)
[21] Misra, J.: Powerlist: a structure for parallel recursion. ACM Trans. Prog. Lang. Syst. 16(6), 1737–1767 (1994)
[22] Musser, D.R.: On proving inductive properties of abstract data types. In: Proc. of POPL '80, pp. 154–162. ACM, New York (1980)
[23] Zhang, H., Kapur, D., Krishnamoorthy, M.S.: A mechanizable induction principle for equational specifications. In: Lusk, E.R., Overbeek, R. (eds.) CADE 1988. LNCS, vol. 310, pp. 162–181. Springer, Heidelberg (1988)

Case-Analysis for Rippling and Inductive Proof

Moa Johansson[1], Lucas Dixon[2], and Alan Bundy[2]

[1] Dipartimento di Informatica, Università degli Studi di Verona
[2] School of Informatics, University of Edinburgh*
moakristin.johansson@univr.it, {l.dixon,a.bundy}@ed.ac.uk

Abstract. Rippling is a heuristic used to guide rewriting and is typically used for inductive theorem proving. We introduce a method to support case-analysis within rippling. Like earlier work, this allows goals containing if-statements to be proved automatically. The new contribution is that our method also supports case-analysis on datatypes. By locating the case-analysis as a step within rippling we also maintain the termination. The work has been implemented in IsaPlanner and used to extend the existing inductive proof method. We evaluate this extended prover on a large set of examples from Isabelle's theory library and from the inductive theorem proving literature. We find that this leads to a significant improvement in the coverage of inductive theorem proving. The main limitations of the extended prover are identified, highlight the need for advances in the treatment of assumptions during rippling and when conjecturing lemmas.

1 Introduction

Inductive proofs are needed to reason about recursion and are thus commonplace in mathematical theories of concern to computer science, such as lists and trees. They are also essential for program verification. In practice, inductive proofs require significant user-guidance and expertise. The theoretical need for this can be seen by the failure, for inductive theories, of cut elimination and decidability [4]. Given the difficulty of automating inductive proofs, it is then sometimes surprising that informal mathematical texts present many proofs simply as "by induction". The ease with which such proofs are informally written is not reflected by automatic, inductive theorem proving. In particular, user-guidance is often needed to specify where case-analysis should be performed.

While earlier inductive proof methods can automatically consider the cases of an if-statement [14,7,19], these systems are first-order and lack pattern matching constructs for inductively defined datatypes. We call these constructs *case-statements*. Higher-order systems can express these constructs and frequently use

* The main part of this research was done while the first author was the the University of Edinburgh. This work was funded by EPSRC grants EPE/005713/1 and EP/P501407/1, as well as grant 2007-9E5KM8 of the Ministerio dell'Università e Ricerca.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 291–306, 2010.

them to define functions. However, in higher-order domains, systems have previously been unable to include the case-analysis steps for pattern-matching constructs within automatic inductive proof search, without sacrificing termination.

The work we describe in this paper improves on the coverage of automatic inductive proof methods by incorporating case-analysis for both if- and case-statements. Our proof technique extends *rippling*, which is a heuristic used for removing the differences between a goal and its inductive hypothesis. It improves on earlier rippling based methods [5] by supporting analysis of case-statements. An important property of our method is that it also preserves termination. This allows many proofs which previously needed explicit user-guidance, or manipulation of the representation, to now be found automatically.

To perform the application of case-analysis within rippling, we introduce a restricted form of resolution which treats variables at the head of a rule as requiring an occurrence of their parameters to be found within the goal. This is then used to guide unification during resolution and avoids the problem that case-analysis rules otherwise unify with every goal.

Our extension of rippling has been implemented in IsaPlanner [9] allowing it to be used for proofs in Isabelle [16]. IsaPlanner did not previously support case-splitting for if- or case-statements. Using our implementation, we tested a large number of examples from both Isabelle's library as well as problems from the inductive theorem proving literature. Our results show that the use of case-analysis in rippling successfully automates the proof of many problems that were previously unprovable by IsaPlanner as well as other systems. Our implementation and more exhaustive details of the experiments are available online[1]. We also analyse the cases where our inductive theorem prover fails. This highlights the need for further work on orthogonal aspects of inductive proof which are no longer limited by case-analysis.

2 Background

2.1 Rippling

Rippling is a heuristic technique for guiding deduction [5]. It is typically used to guide the step cases of inductive proofs. We briefly review the terminology as well as the steps involved by considering a proof of the commutativity of addition.[2] The proof, by induction on the first argument, results in the induction hypothesis $\forall y.\ a + y = y + a$, called the *skeleton*. Rippling annotates the parts of the goal that differ from the skeleton, called *wave-fronts*. For example, in the step-case subgoal, shaded boxes annotate the wave-fronts: $\boxed{Suc\ \boxed{a}}\ + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc\ \boxed{a}}$; where the

[1] http://dream.inf.ed.ac.uk/projects/isaplanner

[2] For the sake of brevity, we do not discuss the various forms of rippling such as static vs dynamic rippling, and we do not need to concern ourselves with details of ripple-measures. The work in this paper is based on dynamic rippling using the sum-of-distances ripple-measure. The interested reader can find further details of such choices in [5,9].

wave-fronts are the two *Suc* symbols. The locations corresponding to universally quantified variables in the skeleton are called *sinks*. In our example, there are two sinks, corresponding to the locations of y, both of which contain b.

An *annotation* for a goal can be constructed from the skeleton and stored separately using *embeddings* [11,18]. For the purposes of this paper, it suffices to consider an embedding simply as way to construct the annotations for a goal. When the skeleton does not embed into the goal, there is no way to annotate the goal such that removing the wave-fronts leaves an instance of the skeleton. Because the correspondence between the skeleton and the goal is lost when there is no way to annotate the goal, such goals are typically considered as worse than those for which there is an embedding.

Informally, one can understand rippling as deduction that tries to move the wave fronts to the top of the term tree, remove them, or move them to the locations of sinks. When all wave-fronts are moved into sinks, or removed, the skeleton can typically be used to prove the goal. This is called *fertilisation*. When the skeleton is an equation, using it to perform substitution in the goal is called *weak fertilisation*. In contrast to this, *strong fertilisation* refers to the case when the goal can be resolved directly with the skeleton. An example proof of the commutativity of addition ending in weak-fertilisation is presented in Fig. 1.

A *rippling measure* defines a well-founded order over the goals, and is constructed from annotated terms. The purpose of a measure is to ensure termination and guide rewriting of the goal to allow fertilisation. Ripple-measures are defined such that, when they are sufficiently low, fertilisation is possible. Each step of rewriting which decreases the ripple measure is called a *ripple-step*.

Definition 1 (Rippling, Ripple-Step). *A ripple step is defined by an inference of the form:*

$$\frac{W,\ s,\ a_2 \vdash g_2}{W,\ s,\ a_1 \vdash g_1} \quad \begin{array}{l} ((t_1 \Rightarrow t_2) \in W) \\ g_1 \equiv t_1\sigma \quad g_2 \equiv t_2\sigma \\ a_1 \in annot(s, g_1) \\ a_2 \in annot(s, g_2) \\ Mes_s(a_2) < Mes_s(a_1) \end{array}$$

The first two conditions identify a rewrite rule in the context W that matches the current goal g_1 and rewrites it to the new goal g_2. The next two conditions ensure that the goals have rippling annotations, a_1 and a_2 respectively, for the skeleton s. The last condition ensures that the ripple measure decrease, where $Mes_s(a_i)$ is the measure with respect to the skeleton s of annotated term a_i. Rippling is the repeated application of ripple-steps.

When there is no rewrite which reduces the measure, either fertilisation is possible, or the goal is said to be *blocked*. The need for a lemma is typically observed when a proof attempt is blocked, or when fertilisation leaves a non-trivial subgoal[3]. There are various heuristics for lemma discovery, the most successful being

[3] By non-trivial, we mean that automatic methods such as simplification cannot prove the subgoal.

$$
\begin{array}{rl}
Suc\ a\ + \lfloor b \rfloor = \lfloor b \rfloor + & Suc\ a \\
\downarrow & \text{Ripple using: } (Suc\ X) + Y = Suc\ (X + Y) \\
Suc(a + \lfloor b \rfloor) = \lfloor b \rfloor + & Suc\ a \\
\downarrow & \text{Ripple using: X + (Suc Y) = Suc (X + Y)} \\
Suc(a + \lfloor b \rfloor) = & Suc(\lfloor b \rfloor + a) \\
\downarrow & \text{Weak fertilisation} \\
Suc\ (b + a) = Suc\ (b + a) &
\end{array}
$$

Fig. 1. A rippling proof for the step-case goal of the commutativity of addition. Ripple-steps move wave-fronts higher up in the term tree and then weak fertilisation is applied.

lemma calculation which applies common subterm generalisation to the goal and attempts to prove the resulting lemma by induction [5,13,9,1].

An important difference between rippling and other rewriting techniques is that the rippling measure is not based on a fixed ordering over terms, but on the relationship between the skeleton, the previous rippling steps, and the goal being considered. This gives rise to two notable features of rippling: its termination is independent of the rules it is given, and, within a single proof, it may apply an equation in both directions. The interested reader can find a more detailed account of rippling in [5,9].

2.2 Isabelle/IsaPlanner

Isabelle is a generic interactive theorem prover which implements a range of object logics, such as higher-order logic (HOL) and Zermelo-Fraenkel set theory, among others [16]. Isabelle follows the LCF-approach to theorem proving, where new theorems can only be obtained from previously proved statements through manipulations by a small set of trusted inference rules. More complex proof methods, called *tactics* are built by combining these basic rules in different ways. This ensures that the resulting proofs rely only on the fixed trusted implementation of the basic inference rules.

Isabelle also has a large theorem library, especially for higher-order logic. The work presented in this paper has been carried out for Isabelle/HOL, although in principle, following Isabelle's design methodology, it can be applied within Isabelle's other object logics. Isabelle/HOL provides powerful definitional tools that allow the expression of datatypes as well as provably terminating functions over them. When the user specifies a datatype, Isabelle automatically derives its defining equations and creates a constant for case-based pattern matching [15]. For example, writing `Nat = 0 | (Suc Nat)` defines the fresh constants $0 :: Nat$ (this is Isabelle's notation for '0 is of type Nat'), and $Suc :: Nat \Rightarrow Nat$ and

derives theorems such as $0 \neq (Suc\ x)$ and $(Suc\ x = Suc\ y) = (x = y)$. A constant $nat\_case : \alpha \Rightarrow (Nat \Rightarrow \alpha) \Rightarrow Nat \Rightarrow \alpha$, is also automatically defined in order to support definition by pattern-matching. When case-constants are applied to their arguments, Isabelle's pretty printing machinery writes them in the more conventional style: `case n of 0 ⇒ c1 | Suc n' ⇒ c2`, where n is the third argument, c_1 is the first, and finally c_2 is the second argument, with n' being c_2's parameter of type Nat. We call such expressions *case statements*.

To facilitate interactive proof, Isabelle has a number of automatic tactics, including a powerful simplification tool which is configured by specifying the set of rules it will apply. The simplification procedure can, for restricted cases, introduce a case-split on the condition of an if-statement; this is discussed further in §6.3. IsaPlanner is a proof-planner for Isabelle [9]. It provides additional abstractions for writing more complex tactics. In particular, an automatic inductive theorem prover based on rippling has been developed in [9,11].

Notation: We will follow Isabelle's notational conventions:

- Theorems with assumptions are written $[\![P;\ Q]\!] \Longrightarrow R$, stating that P and Q are assumptions for the conclusion R.
- Variables that are allowed to be instantiated by unification, are differentiated from those that are not. *Meta-variables* are allowed to be instantiated, and are prefixed by '?', e.g. $?P$.
- The list cons function is written as '#', and we use the '@'-symbol for append.

3 Case-Analysis for Rippling

Isabelle/HOL allows both if-statements, which are directly built into HOL, and case-statements, which are derived for each datatype. Case-statements are more general than if-statements and may introduce new bound variables. Moreover, they provide a convenient way to break datatypes into difference cases without having to introduce well-formedness conditions. More generally, datatypes and their corresponding case-statements are widely used in typed-functional programming. In Isabelle's list library, 16 out of 31 function definitions involve conditional statements, 6 of which use case-statements. Examples include list operations, such as *member* ($\in$) and *delete*, as well as subtraction and $\leq$ for natural numbers. Properties of these functions are typically proved interactively by induction and case-analysis.

Our approach to automate such proofs with rippling is to treat the splitting of conditional statements eagerly – as part of the ripple-step that introduced the conditional statement. After each ripple-step, the case splitting techniques for case- and if-statements are tried. The case- and if-splitting techniques involve two stages: first they attempt to prove that a particular branch will be taken, avoiding the need for a case-analysis. If that fails, then the appropriate case- or if-split is introduced. If the goal does not contain a case- or if-statement, then none of the case-splitting techniques apply and the goal is unchanged. Either way, rippling then continues to try to apply further ripple-steps. The top-level

```
ripple_step = apply_ripple_step THEN ensure_measure_decrease;
rippling = solved OR blocked OR (ripple_step THEN rippling);

ripple_step_with_splits = apply_ripple_step
  THEN (take_if_branch OR split_if
        OR take_case_branch OR split_case OR id)
  THEN ensure_measure_decrease;
rippling_with_splitting = solved OR blocked OR
  (ripple_step_with_splits THEN rippling_with_splitting);
```

Fig. 2. The top-level tactic-style presentation of rippling its extension to include case-analysis for case- and if-statements. The tactic `apply_ripple_step` performs a single step of rippling and succeeds when it satisfies the first three conditions of definition 1 and `ensure_measure_decrease` ensures the last two conditions and tries to prove any non-rippling goals by simplification.

tactic-style script for rippling, and its extension for case-analysis, is shown in Fig. 2 and an illustrative example of its application is presented in § 3.1.

The condition used to ensure that rippling terminates is that each ripple-step decreases the ripple measure. For rippling with case-splits, the ripple-step is modified to include case splitting, and the ripple-measure is checked for each goal after all case-splits are applied. This preserves the termination of rippling, even when performing case analysis on arbitrary datatypes. We discuss, in §3.5, the motivation for eager case-splitting as opposed to considering the introduction and elimination of case- and if-statements as ripple-steps. In §3.3 and §3.4 we give the details of the techniques to handle case- and if-statements respectively.

During a case-split, it is often the case that some branch can no longer be annotated with respect to the skeleton. Such goals are called *non-rippling goals*. Like earlier accounts of conditional-rippling, the ripple-step succeeds when such subgoals can be solved *easily*. In our case, this means by simplification, although other accounts used weaker proof methods (by assumption) [7]. When a non-rippling goal is unsolved by simplification the measure-decrease check fails causing the ripple-step to fail and for search to backtrack. Occasionally, all subgoals after a case-split may be non-rippling goals and are solved by simplification, in which case the `solved` branch of rippling will be taken. Typically, this indicates that the problem did not require proof by induction, but only proof by case-analysis.

3.1 A Simple Example

Below we present a simple example of the application of the case-analysis technique. Due to lack of space, more advanced theorems are available on the web[4].

It is possible to define the *max* function for natural numbers as follows:

$$max\ 0\ y = y \tag{1}$$

$$max\ (Suc\ x)\ y = (case\ y\ of\ 0 \Rightarrow (Suc\ x) \mid (Suc\ z) \Rightarrow Suc\ (max\ x\ z)) \tag{2}$$

[4] http://dream.inf.ed.ac.uk/projects/lemmadiscovery/case_results.php

In an inductive proof of the commutativity of the *max* function, the step-case is:

$$\begin{aligned} &\text{Inductive hypothesis (IH):} && \forall b.\ max\ a\ b = max\ b\ a \\ &\text{Step-case goal:} && max\ \boxed{Suc\ a}\ \lfloor b' \rfloor = max\ \lfloor b' \rfloor\ \boxed{Suc\ a} \end{aligned} \tag{3}$$

By applying rule 2 (`apply_ripple_step`), the left hand side of the step-case is rippled to:

$$\boxed{case\ b'\ of\ 0 \Rightarrow (Suc\ a) \mid (Suc\ z) \Rightarrow Suc(\boxed{max\ a\ \lfloor z \rfloor})}\ = max\ \lfloor b' \rfloor\ \boxed{Suc\ a}$$

At this point, a case-statement has been introduced. Because there are no if-statements, the `take_if_branch` and `split_if` techniques do not apply. The `take_case_branch` also fails as there is no information about the structure of b', as needed to proceed down either branch of the case-statement. The `split_case` technique is then applied. This performs a case-split on b', which allows the proof to proceed and results in the subgoals for the zero and successor cases:

$$b' = 0 \Longrightarrow Suc\ a = max\ b'\ (Suc\ a) \tag{4}$$

$$b' = Suc\ z \Longrightarrow \boxed{Suc(\boxed{max\ a\ \lfloor z \rfloor})}\ = max\ \lfloor b' \rfloor\ \boxed{Suc\ a} \tag{5}$$

Goal 4 cannot be annotated but is solved by Isabelle's simplification tactic (within `ensure_measure_decrease`). Goal 5 is measure-decreasing but then becomes blocked. As discussed in §2 the step-case technique then applies weak-fertilisation and simplification, which in this case completes the proof.

3.2 Applying Case Splits: Restricted Unification in Resolution

To apply a case split, we use a theorem derived for each datatype by Isabelle's definitional machinery. For instance, the following theorem is automatically derived for natural numbers:

$$\begin{aligned} [\![?n = 0 \Longrightarrow ?P(?f_1);\ \forall x.\ (?n = Suc\ x) \Longrightarrow ?P(?f_2\ x)]\!] \Longrightarrow \\ ?P(case\ ?n\ of\ 0 \Rightarrow ?f_1 \mid (Suc\ x) \Rightarrow (?f_2\ x)) \end{aligned} \tag{6}$$

Applied to a case-statement, the meta-variable $?P$ matches the context in which the case-statement occurs. Such theorems allow a case split, to be implemented as a single resolution step. In an interactive setting, the user can specify an instantiation for $?P$. However, in terms of automatic proof by resolution using higher-order unification, the meta-variable $?P$ occurs in head position and thus allows the theorem to be applied to any goal, not just goals that contain case-statements. Furthermore, even when applied to goals containing case-statements, trivial unifiers are found (imitations that throw away the arguments). We want such rules to only find unifiers for goals that contains the meta-variable's argument, which we call the *subterm of interest*.

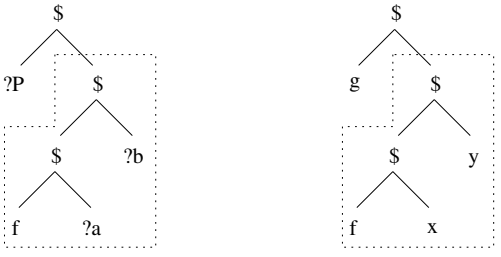


Fig. 3. Examples of zippers with the focus marked by a dashed box. The $-symbol denotes application at the level of the abstract term syntax. [Left] A zipper on the term $?P(f\ ?a\ ?b)$, with focus on the subterm $f\ ?a\ ?b$. [Right] A zipper on $g(f\ x\ y)$, with focus on the subterm $f\ x\ y$.

A simple algorithm, implemented in IsaPlanner, for resolution with such theorems uses a restricted form of unification that first instantiates the head-positioned meta-variable and then performs regular resolution. Our implementation uses *zippers* to traverse the goal term in order to identify the location of the subterm of interest. The zipper maintains the context of this subterm which is used to construct the desired instantiation for the head-variable.

Zippers, as introduced by Huet [12], were motivated by the common problem of needing to represent a tree with a subtree that is the focus of attention. The focus of attention can then be moved left, right, up or down the tree. Figures 3 illustrates zippers over term-trees, where the focus is marked by a dashed box. Using zippers to move around a term has time proportional to the distance moved. Access to the focused subterm and its context is constant time. Importantly, traversal maintains the context. This allows zippers to give programmatic means by which to work on both the subterm of interest and the context in which it occurs.

Below we give an overview of our algorithm for restricted resolution. As an example, assume we have a rule of the form $?P(?a) \Longrightarrow ?P(f\ ?a\ ?b)$ which we wish to resolve with the goal $g(f\ x\ y)$.

1. **Find the argument of the top-level meta-variable:** Check if there indeed is a top-level meta-variable in the conclusion of the rule, otherwise it is safe to proceed with normal resolution. If there is a top-level meta-variable, its argument should match some subterm of the goal that is to be resolved. In our example, the top-level meta-variable, $?P$ in the rule, has the argument $(f\ ?a\ ?b)$. We use a zipper to find this subterm, as shown in Figure 3-left.
2. **Find a matching subterm in the goal:** Using a zipper we traverse the term-tree of the goal until a subterm matching the argument of the meta-variable is found. In the example, $?P$ has one argument, $(f\ ?a\ ?b)$, which matches the subterm $(f\ x\ y)$ in the goal. Figure 3-right shows the zipper of the goal, focused on the unifying subterm.
3. **Instantiate the top-level meta-variable:** The term context surrounding the focused subterm in the goal (everything outside the dashed box in

figure 3-right) is used to construct an instantiation for the rule's top-level meta-variable. The instantiation is created by replacing the focused subterm in the goal with a bound variable and abstracting over it. In our example, this gives the instantiation $?P \equiv \lambda\ z.\ g(z)$.

4. **Resolve with the instantiated theorem:** In our example, resolution is performed with $g(?a) \Longrightarrow g(f\ ?a\ ?b)$, which instantiates the remaining variables, and results in the new sub-goal $g(x)$.

If resolution had been performed without first instantiating $?P$, an extra resulting goal would also be a possibility, namely $(\lambda\ z.\ g(f\ x\ y))\ ?a$ (by imitation in higher-order unification), which reduces the goal to $g(f\ x\ y)$, which is the same as the goal we started with. For rules which have a head-positioned meta-variable in both the conclusion and some assumption, ordinary higher-order unification will find trivial unifiers that result in the same goal as the one that was trying to be proved in the first place. Our technique avoids this problem.

3.3 Case-Statements

As mentioned earlier, each datatype defined in Isabelle has an associated case-constant. This comes with pattern-matching rules for each branch of the case statement. For the datatype of natural numbers these are:

$$(case\ 0\ of\ 0 \Rightarrow ?f_1\ \mid (Suc\ x) \Rightarrow ?f_2\ x) = ?f_1$$
$$(case\ (Suc\ ?n)\ of\ 0 \Rightarrow ?f_1 \mid (Suc\ x) \Rightarrow ?f_2\ x) = ?f_2\ ?n$$

Our case-analysis technique first attempts substitution with one of the above theorems, and if successful it will continue rippling on that branch. If all substitution attempts fail, a case-split is introduced by applying restricted resolution with the appropriate case-splitting theorem. For example, returning to the commutativity of max (§3.1), the step-case subgoal containing the case-statement is:

$$case\ b'\ of\ 0 \Rightarrow (Suc\ a) \mid (Suc\ z) \Rightarrow Suc(max\ a\ \lfloor z \rfloor) \quad = max\ \lfloor b' \rfloor\ Suc\ a \tag{7}$$

Recall the case-split rule for natural numbers (theorem 6, page 297). This can be used to perform a case-split on b' by restricted resolution. This involves first using zippers to partially instantiating $?P \equiv \lambda\ x.\ x = max\ b'\ (Suc\ a)$, and then resolution produces the new 'split' subgoals:

$$b' = 0 \Longrightarrow Suc\ a = max\ b'\ (Suc\ a) \tag{8}$$
$$b' = Suc\ z \Longrightarrow Suc(max\ a\ \lfloor z \rfloor) \quad = max\ \lfloor b' \rfloor\ Suc\ a \tag{9}$$

Observe that following a case-split, an equational assumption, stating the particular value that the case-split term takes, is introduced for each branch. The equation is then substituted in each goal's conclusion. In our example, this means

replacing b' in the conclusions of goals 8 and 9, with 0 and $Suc\ z$ respectively. This is the final step involved in splitting a case-statement into its possible constructor cases. Not performing the substitution complicates further rippling and lemma calculation. For lemma calculation, the substitution frequently removes the need to consider the assumption further, and thus allows one construct more general lemmas. For further rippling, it can cause goals to have no valid annotations or for sinks to contain different, but provably equivalent, terms.

3.4 If-Statements

On encountering an if-statement, our case-analysis technique will first attempt to go down either one of the two branches by substitution using the library theorems:

$$?P \Longrightarrow (if\ ?P\ then\ ?x\ else\ ?y) = ?x \tag{10}$$

$$\neg ?P \Longrightarrow (if\ ?P\ then\ ?x\ else\ ?y) = ?y \tag{11}$$

Applying either of these results in two subgoals. For theorem 10, one subgoal involves proving the condition $?P$ and the other requires proving the then-branch which has substituted the if-statement for $?x$. Similarly, applying theorem 11 involves proving that $?P$ is false, and then proving the else-branch. The subgoal arising from the condition is solved either by resolution with an existing assumption, or by simplification. The other subgoal (the then or else branch) is passed back to rippling.

If the technique fails to show that either the condition $?P$ or its negation holds, a split on the condition is introduced. This is performed by restricted resolution with the library theorem:

$$[\![?Q \Longrightarrow ?P(?y);\ \neg ?Q \Longrightarrow ?P(?z)\ ]\!] \Longrightarrow ?P\ (if\ ?Q\ then\ ?y\ else\ ?z) \tag{12}$$

As before, this results in two new sub-goals. Typically, the skeleton embeds into only one of them, in which case that goal is called the *rippling goal*. Before rippling continues on the rippling goal(s), if there is a non-rippling goal, it is passed to the simplifier and must be solved before rippling continues.

Example. Consider the following theorem: $x \in (l\ @\ m) = x \in l \vee x \in m$. The proof starts by induction on l and then uses the definition of member:

$$x \in (h\#t) = if\ (x = h)\ then\ True\ else\ x \in t$$

Rippling with this rule results in the step-case subgoal:

$$if\ (x = h)\ then\ True\ else\ x \in (l\ @\ m)) \quad = x \in (h\ \#\ l) \quad \vee x \in m$$

The case-analysis technique is then triggered by the discovery of an if-statement in the goal. It is not possible to prove the condition $(x = h)$ or its negation

by simplification, so a split is introduced. Restricted resolution with theorem 12 gives two new subgoals:

$$x = h \Longrightarrow True = x \in (h \# l) \vee x \in m \tag{13}$$

$$x \neq h \Longrightarrow x \in (l \mathbin{@} m) = x \in (h \# l) \vee x \in m \tag{14}$$

The skeleton does not embed into goal 13 and so it is passed to the simplifier, which successfully solves it. Goal 14 is then rippled further by rewriting the right hand side to:

$$x \neq h \Longrightarrow x \in (l \mathbin{@} m) = \mathit{if}\ (x = h)\ \mathit{then}\ True\ \mathit{else}\ x \in l) \vee x \in m$$

This time, taking the *else*-branch succeeds, as the assumption introduced by the previous case-split can be used to show the negation of the condition. The proof is now finished by strong fertilisation.

3.5 Eager Case-Splits

Case-splitting is interleaved with rippling, and applied eagerly whenever a rule introduces a case- or if-statement. The rule introducing the case statement, followed by application of the case-split itself, is regarded as a single ripple-step. This has two main advantages over waiting until rippling is blocked, or including the case-split as a separate rule in rippling, as in previous approaches [5].

Firstly, some ripple measures are not reduced between the goal containing a case-statement and the resulting goal after the split. In the example proof of goal 7, the case-statement has a wave-front in the same position, with respect the skeleton, as the goal (9) after the split. Ripple measures which are invariant on the size of wave fronts hence filter out such steps as they are not measure decreasing. By treating a rule's application and the following case-split as a single ripple-step, all known ripple-measures decrease with respect to the previous goal (3). Similarly, for splitting data-types, substitution with the introduced case-assumption is typically not measure decreasing and hence needs to be included as part of the compound ripple-step.

Secondly, when case- and if-statements can be reduced to a known branch, such that an actual case-split is not required, our technique proceeds directly down the relevant branch. If the case- or if-statement is allowed to remain in the goal, redundant rippling steps might be applied to a branch which is later discarded. Eager application of the case-splits is thus more efficient on such problems.

4 Evaluation

Functions defined using if- and case-statement are very common, but many proofs requiring the corresponding case- analysis could not previously be found by rippling based methods. Rippling with the case-analysis technique has been

evaluated, in IsaPlanner, on a set of 87 theorems involving lists, natural numbers and binary trees. These are defined using if- or case-statements, none of which IsaPlanner could prove previously. 47 of the theorems can now be proved automatically using our case-analysis technique. Of the theorems in this evaluation corpus, 41 of the proofs involve if-statements, 41 involve case-statements and 5 involve both. Most of the theorems in the corpus are a subset of inductive theorems from Isabelle's libraries for lists and natural numbers[5]. Some are more programmatic in character and taken from the CLAM system [13] and from problems arising from dependently typed programming [20]. The criteria used to select the theorems was simply that they require inductive proof and involve some function(s) defined using if- or case-statements. We also added some further theorems to check that our machinery worked with other common properties and definitions. The evaluation corpus and full results, including the run-times are available on-line[6].

We did not expect IsaPlanner to prove all theorems even with the new case-analysis technique. Many of the remaining 40 theorems require, in addition to case-analysis, support for generating conditional lemmas or more elaborate reasoning about side-conditions than IsaPlanner currently is capable of. These theorems are included in the corpus to identify areas for further development of the prover. With case-analysis techniques now available, we propose further extensions to IsaPlanner in §5.

The theorems were proved only from function definitions, rippling was not provided with any extra lemmas. The experiments were run on an Intel 2 GHz processor. All proofs were found in less than one second. Some failed proofs took slightly longer, with the maximum of 9 seconds for one proof attempt. For the experiments we used IsaPlanner's rippling-based inductive prover [9], which has been extended with our case-analysis technique. We also compared this prover with one that applies induction followed by Isabelle's simplifier and lemma calculation [10]. The simplifier applies rewriting with the definitions from left to right which ensures termination. However, lemma calculation can lead to infinite chains of conjectured lemmas. The results of the comparison show that the simplification-based prover differs from the rippling-based one in two important respects:

Proved Theorems: The simplification-based prover managed to prove 37 of the 87 theorems. There are 15 theorems rippling can solve but simplification cannot, most of these require a split on a case-statement, which simplification is unable to perform. In general case-splitting for datatypes makes simplification non-terminating. There are also 6 theorems simplification proves but rippling fails to prove. These involve more sophisticated reasoning with assumptions than rippling currently employs. Interestingly, when the standard set of simplification rules from Isabelle's library are available, rippling's performance improves more than the simplification-based technique; there are then 20 theorems provable by

[5] isabelle.in.tum.de/dist/library/HOL/index.html

[6] http://dream.inf.ed.ac.uk/projects/lemmadiscovery/case_results.php

the rippling prover and but not by the simplification-based one. The number of theorems that simplification can prove, but rippling cannot, is unchanged.

Termination and Conjecturing: On problems that it cannot solve, the simplification based prover fails to terminate. In contrast, the rippling based prover terminates on all proofs. When asked for alternative proofs, rippling also maintains termination while the simplification based prover again fails to terminate. When analysed in more detail, we observed that conjecturing after simplification frequently leads to attempting to prove an infinite chain of increasingly complicated conjectures. Rippling, on the other hand, does not suffer from this problem. We conclude that the heuristic guidance of rippling leads to better lemmas being calculated.

5 Further Work

The implementation of techniques for case-analysis have increased the power of IsaPlanner's inductive prover. The failed proofs in the evaluation set highlight a number of areas for further work that are orthogonal but complementary to case-splitting. Firstly, this paper has not focused on lemma discovery. An interesting future direction is to explore automated discovery of conditional lemmas, which are needed in many proofs. An example is the proof that insertion sort produces a sorted list: $sorted(insertion\_sort\ l)$. IsaPlanner's current lemma discovery techniques are limited to making conjectures without assumptions. However, in this case the needed lemma is $sorted\ m \Longrightarrow sorted(insert\ x\ m)$.

Secondly, extensions to improve IsaPlanner's capabilities to reason about more complex conditional conjectures, and about goals with multiple assumptions would further increase the power of the prover. Such goals occur more frequently in domains where case-splitting is applied, introducing new assumptions. Implementing extensions to fertilisation, as described in [2] may prove beneficial in these cases. An example where this would be useful is in the proof of the lemma that needs to be conjectured to prove the correctness of sorting: $sorted\ m \Longrightarrow sorted(insert\ x\ m)$. The step case would be solved by rippling forward to prove $sorted\ (h\#t) \Longrightarrow sorted\ t$, and rippling backward to prove $sorted\ (insert\ x\ t) \Longrightarrow sorted\ (insert\ x\ (h\#t))$. This requires both rippling forward from assumptions and, as is currently implemented, backwards from the goal, respectively.

Finally, for this paper, we have not been concerned with the issue of induction scheme selection. IsaPlanner's induction tactic does allow the user to specify a custom induction scheme, but when running entirely automatic, as in our experiments, IsaPlanner's default setting is to use structural induction on a single variable. However, with such an induction scheme, some proofs will then require many nested case-splits, interleaved with new inductions. As there is a risk of non-termination, IsaPlanner only allows new inductions in the context of lemma speculation, and not for non-rippling goals arising after case-splits. Exploring heuristics for automatically selecting and applying induction schemes over multiple variables simultaneously will enable full automation in more of these cases.

An example where this is beneficial is the proof of right-commutativity of subtraction, expressed in Isabelle's library as *(i - j) - k = (i - k) - j*. However, even when more elaborate induction schemes are used, there is no guarantee that case-statements may not be introduced at some intermediate point in the proof, perhaps from rewrite rules arising from other conditional functions, or from auxiliary lemmas. Therefore, the case-analysis technique will still be of use.

6 Related Work

6.1 Induction-Scheme Based Case-Analysis

Approaches to selecting or deriving induction schemes, such as recursion analysis [3], can avoid the need to perform additional case-splits. This is how systems such as ACL2 [14] and VeriFun [19] tackle problems that otherwise need case-analysis. As these system do not have datatypes, unlike Isabelle/IsaPlanner, all recursive functions are defined using if-statements and destructor constructs. For example, append is defined by: *x @ y = if x=[] then y else hd(x)#(tl(x) @ y)*. Functions involving case-splits on multiple variables are defined by recursion on several arguments, and hence recursion analysis is able to construct the appropriate induction scheme.

In Isabelle, definitions are typically constructor style. While and proofs could be translated into destructor style to avoid dealing with case-statements, this would require translating all function definitions and auxiliary lemmas as well. However, the problem of how to control the unfolding of case-statements would not be solved; instead it would become a problem of conditional rewriting where new variables are introduced. Furthermore, translation would produce longer, less readable proofs. Another example when recursion analysis over destructor style definitions is not appropriate is during function synthesis, where new forms of recursion need to be derived [6]. Curiously, using case-analysis, as opposed to selecting and constructing richer induction schemes also sometimes avoids introducing unnecessary base-cases.

6.2 Case-Splitting in Other Rippling Based Provers

In version 3 of the CLAM system [8], conditional functions would typically be defined using several conditional rewrite rules. For example, *member* would, in the non-empty case, be written using the rules: $x = h \Longrightarrow x \in (h \# t) = True$ and $x \neq h \Longrightarrow x \in (h \# t) = x \in t$. If one of these is applicable but the condition cannot be proved by simplification, and there exists another rule with a complementary condition, CLAM's case-split critic is triggered and introduces a split on the condition [13]. In Isabelle/IsaPlanner, functions with conditions are typically defined using an if- or case-statement. This is why our case-analysis technique works on if- and case-statements, rather than complementary conditional rewrite rules as in CLAM. As CLAM works only in first-order domains, it does not include case-statements and its case-analysis critic cannot perform the corresponding splits on datatypes. Of the 87 theorems in our evaluation

corpus, CLAM could thus not have proved any of the 41 theorems about functions defined using case-statements. The λCLAM system [17], while being able to express case-statements, had no proof methods working with them.

6.3 Simplification

Simplification based tools, such as Isabelle's simplifier, can automatically split if-statements, but not case-statements ([15], §3.1.9). In general, splitting case-statements might cause non-termination for rewriting. The user is therefore required to identify and insert case-splits where required, or apply a more sophisticated induction scheme, such as simultaneous induction on several variables. Our technique, on the other hand is incorporated as a step within rippling and the rippling measure ensures termination. Splitting case-statements is safe as long as the ripple-measure decreases. IsaPlanner employs the simple default structural induction scheme for the datatype. Thanks to the case-analysis technique, IsaPlanner still succeeds in automatically proving theorems such as *(i - j) - k = i - (j + k)*, which in the interactive proof from Isabelle's library uses a custom induction scheme chosen by the user.

7 Conclusions

Performing case-splits is an important feature for an automatic inductive theorem prover. It is needed to prove properties of many functions that are naturally defined using if- and case-statements. Our case-analysis technique can perform the needed case-splits for many of these cases. It is triggered during rippling whenever an if- or case-construct is encountered. If it is possible to prove the associated condition, the technique proceeds down the corresponding branch, otherwise it introduces a split. Performing such case splits automatically by naive resolution is applicable to all goals. By introducing a restricted form of resolution we were able to take advantage of the automatically derived library theorems. The technique has been fully implemented and tested in IsaPlanner. It is incorporated with rippling, which ensures termination. Our evaluation showed that 47 out of 87 theorems which required case-analysis could be prove automatically by IsaPlanner. Splitting the cases of a pattern matching statement was needed for 14 of these problems. Other forms of rewriting such as Isabelle's simplifier, are non terminating in these cases. More difficult conditional theorems from our evaluation corpus require the capability to conjecture conditional lemmas and improved reasoning with assumptions, which we suggest as future work.

References

1. Aderhold, M.: Improvements in formula generalization. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 231–246. Springer, Heidelberg (2007)
2. Armando, A., Smaill, A., Green, I.: Automatic synthesis of recursive programs: The proof-planning paradigm. Automated Software Engineering 6(4), 329–356 (1999)

3. Boyer, R.S., Moore, J.S.: A Computational Logic. ACM monograph series. Academic Press, London (1979)
4. Bundy, A.: The automation of proof by mathematical induction. In: Handbook of Automated Reasoning, vol. 1. Elsevier, Amsterdam (2001)
5. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-level Guidance for Mathematical Reasoning. Cambridge Tracts in Theoretical Computer Science, vol. 56. Cambridge University Press, Cambridge (2005)
6. Bundy, A., Dixon, L., Gow, J., Fleuriot, J.: Constructing induction rules for deductive synthesis proofs. In: CLASE at ETAPS-8. ENTCS, vol. 153, pp. 3–21 (2006)
7. Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., Smaill, A.: Rippling: A heuristic for guiding inductive proofs. Artificial Intelligence 62(2), 185–253 (1993)
8. Bundy, A., van Harmelen, F., Horn, C., Smaill, A.: The Oyster-Clam system. In: Stickel, M.E. (ed.) CADE 1990. LNCS (LNAI), vol. 449, pp. 647–648. Springer, Heidelberg (1990)
9. Dixon, L.: A Proof Planning Framework for Isabelle. PhD thesis, University of Edinburgh (2006)
10. Dixon, L., Fleuriot, J.D.: IsaPlanner: A prototype proof planner in Isabelle. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 279–283. Springer, Heidelberg (2003)
11. Dixon, L., Fleuriot, J.D.: Higher order rippling in IsaPlanner. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 83–98. Springer, Heidelberg (2004)
12. Huet, G.: The zipper. Journal of Functional Programming 7(5), 549–554 (1997)
13. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. JAR 16(1-2), 79–111 (1996)
14. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common Lisp. IEEE Transactions on Software Engineering 23(4), 203–213 (1997)
15. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Paulson, L.C.: Isabelle: A generic theorem prover. Springer, Heidelberg (1994)
17. Richardson, J.D.C., Smaill, A., Green, I.: System description: proof planning in higher-order logic with Lambda-Clam. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 129–133. Springer, Heidelberg (1998)
18. Smaill, A., Green, I.: Higher-order annotated terms for proof search. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 399–413. Springer, Heidelberg (1996)
19. Walther, C., Schweitzer, S.: About VeriFun. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 322–327. Springer, Heidelberg (2003)
20. Wilson, S., Fleuriot, J., Smaill, A.: Automation for dependently typed functional programming. Special Issue on Dependently Typed Programming of Fundamental Informaticae (2009)

Importing HOL Light into Coq

Chantal Keller[1,2] and Benjamin Werner[2]

[1] ENS Lyon
[2] INRIA Saclay–Île-de-France at
École polytechnique
Laboratoire d'informatique (LIX)
91128 Palaiseau Cedex
France
{keller,werner}@lix.polytechnique.fr

Abstract. We present a new scheme to translate mathematical developments from HOL Light to Coq, where they can be re-used and re-checked. By relying on a carefully chosen embedding of Higher-Order Logic into Type Theory, we try to avoid some pitfalls of inter-operation between proof systems. In particular, our translation keeps the mathematical statements intelligible. This translation has been implemented and allows the importation of the HOL Light basic library into Coq.

1 Introduction

1.1 The Curse of Babel?

Proof-systems are software dedicated to the development of mechanically checked formal mathematics. Each such system comes with its own logical formalism, its own mathematical language, its own proof language and proof format, its own libraries. A consequence is that it is largely impossible to reuse a formal development of one system in another, at least not without a re-implementation requiring important amounts of human work and skills.

This situation is about as old as proof-systems themselves, has often been deplored and is mostly felt as a modern form of the curse of Babel.

On the other hand, if the large number of human languages sometimes hinders comprehension between people, it also gives them a broader set of means to express themselves. It is well known that many subtleties of an original text are often "lost in translation". A similar point can be made in the case of formal mathematics: certain formalisms and systems can allow smoother developments of, say, real analysis, while others will be more at ease with fields involving large combinatorial case analyzes.

For these reasons, automatic translation between proof-systems is a tempting and old idea. It has, however, been hindered by various theoretical and practical obstacles. We here describe a new attempt that opens new possibilities. The ideas underlying this work are:

– We focus on one particular case, the translation from HOL Light to Coq.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 307–322, 2010.

– This work is specific to this case, and builds upon a careful study of the logical formalisms of both systems and the way they are implemented.
– In this particular case, we provide a *good* translation, in the following sense: the statements of the theorems translated from HOL Light remain intelligible and can be incorporated into further developments made in Coq.

1.2 Embedding Higher-Order Logic into Coq

As it is often the case when logic meets implementation, there are two aspects in this work:

1. The choice of logical embedding: in our case, statements and proofs of HOL Light have to be translated into counterparts in Coq's type theory. For instance, one often distinguishes between *deep* and *shallow* embeddings. The choice of this translation is central and will be discussed below.
2. The way to actually implement this translation. This will depend on issues like the way the two systems represent proofs, whether the translating function processes proof objects or proof scripts, etc...

Deep and shallow embeddings. In order to represent terms from one logical framework A inside another formalism B, we have two possible ways:

– A *deep* embedding: define data-types in B that represent types and terms of A; we can then define, inside B, what it means to be provable in A.
– A *shallow* embedding: represent types and terms of A using their counterparts in B; this translation must preserve provability.

A deep embedding can ease things on the implementation side: we have access to the structure of the terms, and we can reason about them. Furthermore, the data types serve as a well-defined interface between the two systems.

However, our ultimate aim is to obtain actual Coq theorems[1]. For that, we concretely need a shallow embedding. In a previous work by F. Wiedijk [20], theorems from HOL Light were directly translated through a shallow encoding. Wiedijk observed that automation was difficult to perform that way. Furthermore, and although he used a shallow embedding, the theorems he obtained were still somewhat awkward in Coq. For instance, we would like to see the theorem $\forall a, \exists b, a = b$ to be translated in Coq in `forall (A:Type)(a:A), exists b, a = b` whereas Wiedijk obtains `forall (A:hol_type), hol_thm (hol''forall A (hol_Abs A hol''bool (fun (a:A)=> hol''exists A (hol_Abs hol''bool (fun (x:A)=> hol''eq A x a)))))`.

To avoid these pitfalls, we are going to obtain this shallow encoding going through a deep one. Frameworks like type theories allow to lift a deep embedding into a shallow one. This operation is called *reflection* from a proof theoretic point

[1] If John Harrison provides us with, say, Wiles' proof of Fermat's theorem in HOL Light, we want to translate it to the "real" Coq formulation of this famous result, and not the theorem "Fermat's theorem as stated in HOL is provable in HOL".

of view; through a proofs-as-programs perspective it can be understood as a compilation operation and corresponds to one of the two steps of *normalization by evaluation* (NbE) [7]. In this work we adapt a previous formalization of NbE in Coq [12].

To conclude, from HOL Light, we get deeply embedded terms; then we translate them into Coq theorems using a process similar to the computation part of normalization by evaluation.

Motivations and difficulties. Embedding the Higher-Order Logic into Coq means defining a model of this logical framework, and so proving its *coherence*. The confidence we can have about the HOL Light theorem prover is thus increased. And this is enforced by the ability we have to check in Coq the theorems that were proved in HOL Light.

When translating theorems from HOL Light to Coq, we will have to take into account the differences between the two systems: as they often make different choices in the way a same mathematical corpus is formalized, one can therefore fear that a translated theorem is difficult to reuse in the target system, or that its statement becomes obscured.

Finally, one expects such a translation to be robust to changes in both proof-systems, and to be as efficient as possible in terms of time and memory consumption.

1.3 Related Work

Interaction between HOL and Coq. There already have been some attempts to share proofs between HOL systems and Coq, some of which are being developed now. One solution is to rely on an independent tool that will, at the same time, check the proof and perform the translation [9,2]. Closer to our approach, it is conceivable to transform a theorem in the HOL system into a theorem in Coq, and check it in Coq, using its HOL proof as a guideline. F. Wiedijk [20] observed that directly using a shallow encoding was hard to automate and resulted in rather unintelligible theorems in Coq.

Recording and exporting HOL proofs. Efficient systems to record and export HOL proofs have already been developed, both for the HOL prover [21] and for HOL Light [19,17]. These works pursue the same motivations as ours: to import HOL proofs into a theorem prover. Since Obua's tool [19] is far more stable and easier for a direct exportation to Coq, we reused his code on the HOL Light side (proof recording), and changed the exportation side to a Coq format. These changes are now distributed with the development version of HOL Light[2].

1.4 Outlines

In the next section, we see the main characteristics of HOL Light and Coq that serve our work. Section 3 is dedicated to the major part of our work: constructing

[2] The development version of HOL Light is currently available at `http://hol-light.googlecode.com/svn/trunk`

a deep embedding of the HOL Light logic into Coq, that is defining data structures that represent types, terms and derivations of HOL Light. To relate this to our main goal, we have to work on two levels:

- Upstream: we record and export HOL Light proofs into this encoding.
- Downstream: inside Coq, we construct a *lifting* from the deep embedding to a shallow embedding. Each HOL Light construction is mapped to its Coq counterpart. Besides, we prove that each time we have a Coq object representing a correct HOL Light derivation, its translation is a correct Coq theorem (Section 4).

This way, from a proof tree imported from HOL Light, we can reconstruct a theorem statement and its proof, using reflection.

We develop some aspects of the way to obtain reasonable performances in Section 5, thus obtaining the results described in Section 6. Finally, we discuss our approach and its perspectives.

2 HOL Light and Coq

HOL Light [15] and Coq [6] are two interactive theorem provers written in OCaml [18]. Although the ancestries of Coq and HOL Light can both be traced back to LCF, there are important differences, between the logical formalisms as well as in the way they are implemented. For obvious matters of space, we here do not give a complete description of the two systems, but underline differences which are crucial for the translation.

A more detailed comparison between HOL [14] and Coq has been established in [22] and enhanced in [9]. These studies also apply to HOL Light, which mainly differs from HOL by its smaller implementation.

2.1 The Status of Proofs

Proof systems like Coq and HOL Light share the same goal: to construct a formal proof. However, the status of these constructions is different in the two systems. This difference is directly related to the two formalisms.

HOL Light implements a variant of Church's Higher-Order Logic. A proof is a derivation in natural deduction. In the example below, Γ and Δ are sets of hypotheses; s, t, u and x are objects of some well chosen types:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \text{TRANS} \qquad \dfrac{}{\vdash x = x}\ \text{REFL}}{\Gamma \cup \Delta \vdash s(x) = u(x)}\ \text{MK\_COMB}}$$

While such a derivation has an obvious tree structure, it is not constructed as such in the system's memory. HOL Light represents statements like $\Gamma \vdash s = t$ as objects of an ML abstract data type `thm`. The fact that this data type is abstract

is the key to the system's safety[3]: the only way to construct objects of type `thm` are well-understood *primitive tactics* corresponding to primitive inference rules. Thus, if a statement $\Gamma \vdash P$ can be represented in `thm`, it *is indeed* a theorem.

Coq implements a *Type Theory* where proofs are objects of the formalism[4]. More precisely, "being a proof of P" is identical with "being of type P", and as a consequence, the statements of the theory are of the form $\Gamma \vdash t : P$.

This results in a very different architecture and safety model:

- The type-checker is the safety critical part of the system, its *trusted computing base*.
- Theorems are constants: the statement is the type and the body is the proof. Proofs are thus kept in the system's memory and can be re-checked.

Consequences. Since Coq keeps proof-objects while HOL Light does not, we need to build these objects at some point in the translation process. The translated proofs can have a critical size. We will see below that this requires special care.

2.2 Computations and Equality

In both formalisms, the objects are strongly normalizing typed λ-terms. The way normalization is performed is very different however. Coq allows to type actual functional programs: its objects include a purely functional and terminating kernel of ML. This leads to some computation over terms, and a notion of *convertibility* between β-equivalent terms. In HOL Light, no computation is performed, and two β-equivalent terms are only *provably equal*.

Let us take an example.

In Coq, addition over unary natural numbers is defined as the usual program:

```
Fixpoint plus (n m : nat) : nat := match n
  with | O => m | S p => S (plus p m) end.
```

Such programs come with a notion of computation, which defines a notion of intentional equality. In this case, extended β-reduction yields (with some syntactic sugar) $2 + 2 \rhd_\beta 4$ and thus $2 + 2 =_c 4$.

Like in all Martin-Lf type theories, these, possibly complex, computations are taken into account in the formalism by the specific *conversion rule*:

$$(\textsc{Conv})\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ (\text{if } A =_c B)$$

As a consequence, computations are omitted in the proof; for instance the propositions $\text{even}(2 + 2)$ and $\text{even}(4)$ have exactly the same proofs.

Another consequence is that Coq and similar type theories need a notion of *propositional equality* which is distinct from the *computational relation* $=_c$.

[3] And to the safety of related systems (LCF, HOL).

[4] This is one aspect of the *Curry-Howard isomorphism*; another aspect is that the Type Theory is constructive, but this point is less central here.

In HOL Light, on the other hand, one will *prove* the existence of addition, that is of a function verifying the properties $0 + x = x$ and $S(x) + y = S(x + y)$. The equality predicate is the way to construct atomic propositions, and all constructions ($\wedge, \vee, \Rightarrow$ connectors, quantifiers...) are defined through combinations of equality and, in some cases, the ε-operator.

HOL Light's equality has specific rules corresponding to reflexivity, transitivity, extentionality, β and η-equivalence.

Consequences. On the one hand, in Coq, the length of a proof can be reduced by maximizing the computational part, which does not appear in proofs. This principle is called *computational reflection*. In our development, proofs of the theorems that are imported from HOL Light use these computational capabilities of Coq.

On the other hand, in HOL Light, the absence of computation leads to big proofs. The advantage is that we avoid the implementation of a β-reduction over our representation of HOL Light's λ-terms and a conversion rule, really simplifying *de facto* proof checking.

2.3 Treatment of Constants

Constants, that is the ability to have definitions, are essential to any mathematical development. Precisely because of the different statuses of equality, constants are treated differently in the two systems.

In Coq, constants are treated through the computational equality and the conversion rule. Whenever a constant is defined:

```
Definition c : A := body.
```

a new object `c` of type `A` is added to the environment, and the computational equality is extended by $\mathtt{c} =_c \mathtt{body}$.

In HOL Light, constant unfolding is explicitly handled by the equality predicate. The corresponding definition will also yield an object `c` of type `A`, together with a *theorem* stating that $\mathtt{c} = \mathtt{body}$.

Consequences. The fact that constants in HOL Light are unfolded only by invoking a specific theorem will turn out to be very convenient: it will allow us to map specific constants to previously defined Coq objects (see Section 4.3).

2.4 Classical Logic

Whereas **Coq**'s intentional type theory is constructive, **HOL Light**'s logic is:

- Extensional: the η-equivalence is assumed (and functional extensionality can be deduced from it in HOL Light).
- Classical: propositions are actually boolean terms; thus propositional extensionality is true by definition; furthermore the axiom of choice is assumed via the introduction of a primitive Hilbert ε-operator.

Consequences. Given the fact that the classical reasoning is deeply embedded in HOL Light logic, there seems to be no alternative but to assume corresponding axioms in Coq (excluded middle, choice operator, extentionality). Indeed, we need to apply them when proving the coherence of HOL Light in Coq.

2.5 Type Definitions

In Coq, type definitions are just regular definitions or inductive definitions. **In HOL**, there is no primitive notion of induction, and since types have a different status than terms, type definitions have to be handled specifically.

A user can define new types in different ways involving facilities such as induction, but they all rely on one mechanism: the schema of specification. This is implemented by the primitive rule `new_basic_type_definition`, which, given a property $P : A \to \text{bool}$, a term $x : A$ such that $\vdash P\ x$ holds, defines the type $B = \{y : A | P\ y\}$, and two constants (in the sense of section 2.3): the canonical injection from B to A and the injection from A to B whose behavior is specified only for elements y such that $P\ y$ holds.

Consequences. All the types in HOL Light are inhabited, since the base types are inhabited and a type definition has to be inhabited because we require $\vdash P\ x$. This is useful to define the ε operator in a proper way without a condition of non-emptiness on the type.

3 Deep Embedding: Representing Higher-Order Logic in Coq

In this section, we represent Higher-Order Logic in Coq. The main originality is the computational definition of deduction at the end of the section. The basic definitions are quite standard, but some care is needed in order to make the later developments tractable and to keep memory consumption as low as possible as well as the computations efficient enough. In particular, it is mandatory to have explicit definitions.

Our encoding uses the SSReflect tactics and libraries package developed by G. Gonthier et al. [13], but the level of details of this paper does not allow us to make clear how crucial this is.

3.1 Types and Terms

Names. We need names for variables and definitions both for types and terms. These names have to be in an infinite countable set over which equality is decidable. For efficiency reasons, we chose `positive`, the Coq representation of binary natural numbers greater than 1.

In the rest of the paper, `idT`, `defT`, `idV` and `defV`, respectively representing types variables, types constants, terms variables and terms constants, stand for `positive`.

Types. For types, we need variables and definitions, and we give a specific status to the two primitive definitions of HOL Light: `bool` and the functional arrow. A previously defined type is accessed through its name and the list of its arguments.

```
Inductive type : Type :=
| TVar : idT → type | Bool : type
| Arrow : type → type → type
| TDef : defT → list_type → type
with list_type : Type :=
| Tnil : list_type
| Tcons : type → list_type → list_type.
```

Terms. Terms are defined straightforwardly as an inductive type. For bound variables, we use a locally nameless representation [5], which is simpler to reason about than a named representation. As for types, we distinguish some primitive term definitions: the equality, the ε choice operator, and the logical connectives. We group them together under the type `cst`, and obtain for terms this definition:

```
Inductive term : Type :=
| Dbr : nat → term | Var : idV → type → term
| Cst : cst → term | Def : defV → type → term
| App : term → term → term
| Abs : type → term → term.
```

Typing. This is our first use of computational reflection. Rather than defining typing judgments as an inductive relation, we directly define [12] a typing algorithm. Given a De Bruijn `context` (a list of types) `g` and a term `t`, the function `infer` returns the type of `t` under the context `g` if it exists, and fails otherwise. `wt g t A` means that this term has type `A` under `g`, and is just a shortcut for "`infer g t` returns `A`". Since we consider simply typed terms, the definition of `infer` is easy.

3.2 Derivations

We now define HOL Light's logical framework. Typically, Coq's induction process and dependant types are well-suited to represent such judgments. Here is an extract of the inductive data-type that represents HOL Light derivations):

```
Inductive deriv : hyp_set → term → Prop :=
| Drefl : forall t A, wt nil t A → deriv
   hyp_empty (heq A t t)
| Dconj: forall h1 h2 a b, deriv h1 a → deriv
   h2 b → deriv (hyp_union h1 h2) (a hand b)
| Dconj1: forall h a b, deriv h (a hand b) →
   deriv h a
| Dconj2: forall h a b, deriv h (a hand b) →
   deriv h b
| ...
```

It would however be impractical to have HOL Light generate such derivations. They would be too verbose and difficult to build (for `Drefl`, how to prove `wt nil t A`?). Obua [19] notices that it is sufficient to build a far more compact *skeleton* of the tree. This skeleton carries only minimal information; typically which inference rules have been used.

Following his code, we record proofs in HOL Light using an ML recursive type that represents this skeleton (which means this structure uses no dependent types anymore). We then export it straightforwardly into its twin Coq inductive type:

```
Inductive proof : Type :=
| Prefl : term → proof
| Pconj : proof → proof → proof
| Pconjunct1 : proof → proof
| Pconjunct2 : proof → proof
| ...
```

These twin OCaml and Coq types thus establish the bridge between HOL Light and Coq.

This structure is typed too loosely to guarantee the correctness of the derivation. Some objects of type `proof` do not correspond to actual proofs. For instance, `Pconjunct1` stands for the elimination rule of $\wedge$ on the left. Its argument should be a proof of a theorem that looks like $\Gamma \vdash A \wedge B$, but this is not enforced in the `proof` inductive type.

However, a skeleton is sufficient to reconstruct a complete derivation when it exists, by making the necessary verifications. This is the job of the `proof2deriv` function:

- If `p` is a well-formed proof, then `proof2deriv p` returns `h` and `t`, and we can establish that `deriv h t` stands (this is lemma `proof2deriv_correct`).
- If not, `proof2deriv` fails.

Once more computational reflection is used to deduce `deriv h t` from a `proof`.

4 Going from the Deep Embedding to Coq Terms

In the previous section, we show how to export HOL Light proofs and obtain in Coq objects of type `deriv h t` for a certain `h` and `t`, that is to say deeply written theorems. To interact with Coq theorems, we want to have a shallow reading of these theorems. That is why we define a translation from deep to shallow.

This translation has already been implemented in [12] for a simply typed λ-calculus with only named variables. Here it is trickier with De Bruijn indices and definitions.

4.1 General Idea

For the moment, we suppose given an interpretation function $\mathcal{I}$ to interpret variables and definitions names. Its meaning is detailed in Section 4.3.

We first define $[\bullet]_{\mathcal{I}}$, a translation function on types, that maps, for instance, `Bool` to `Prop` and the arrow to the arrow of Coq (see its precise typing below). We then define $|\bullet|_{\mathcal{I}}$, a translation function on terms, that respects their types. Informally, it means that when a term t has type T, then $|t|_{\mathcal{I}}$ belongs to $[T]_{\mathcal{I}}$:

$$\text{if } \Gamma \vdash t : T \text{ then } |t|_{\mathcal{I}} \in [T]_{\mathcal{I}}$$

4.2 Implementation

It is important for all the types in HOL Light to be inhabited, as we noticed in Section 2.5. We now cope with this by stating that the translation of a type must be an inhabited `Type`. Inhabited `Type`s can be represented by a record:

```
Record type_translation : Type :=
  mkTT {ttrans :> Type; tinhab : ttrans}.
```

The translation function $[\bullet]_{\mathcal{I}}$, a.k.a. `tr_type`, maps a HOL type to a `type_translation`:

```
tr_type: forall I, type → type_translation
```

The translation function $|\bullet|_{\mathcal{I}}$, a.k.a. `sem_term`, is as a refinement of the typing function `infer` (see Section 3.1). In addition to typing a term, it gives its Coq translation (note that its definition is eased by the use of dependant types):

```
sem_term : context → term →
  option {ty: type & forall I, tr_type I ty}
```

4.3 The Interpretation Functions

Variables. We have three kinds of variables to interpret: type variables, named term variables and De Bruijn indices. We map each of them to Coq objects with respect to their types.

Definitions. The interpretation of definitions is a key point to preserve the intelligibility of theorems. Interpretation for types and terms definitions are respectively defined by:

```
Definition type_definition_tr := defT →
  list type_translation → type_translation.
Definition term_definition_tr :=
  defV → forall (A: type), tr_type I A.
```

Imagine we have a HOL Light term `t` bringing into play objects of type `num`, the type for natural numbers defined with zero (`_0`) and successor (`SUC`). If we apply `sem_term` to `t` with an object of type `type_definition_tr` that maps `(num,[])` to `nat` and a object of type `term_definition_tr` that maps `(_0,num)` to `0` and `(SUC,num→num)` to `S`, we obtain a Coq term that corresponds to `t`, but it is a standard Coq theorem, and it would have been written that way directly in Coq.

Besides, this process is fully *flexible*: if instead of `nat`, we map `num` to `N`, the Coq type for binary natural numbers, then we get the same term in this different formalism of naturals.

Notation. In the remaining of the article, the interpretation functions are still abbreviated as $\mathcal{I}$.

4.4 Adequacy of Derivations w.r.t. Semantics

We can now establish that this translation applied to correct HOL Light derivations produce correct Coq theorems. By a correct Coq theorem, we mean a term that is locally closed, has type `Bool`, and whose translation is a correct Coq proposition whatever the interpretation functions might be:

```
Definition has_sem (t: term) : Prop :=
  match sem_term nil t with
    | Some (existT Bool evT) ⇒ forall I, evT I
    | _ ⇒ False end.
```

We can establish the following theorem:

```
Theorem sem_deriv : forall (h: hyp_set) (t:
   term), deriv h t → forall I, sem_hyp I h →
   has_sem I t.
```

where `sem_hyp` $\mathcal{I}$ `h` checks that `has_sem` $\mathcal{I}$ holds for each term of `h`. Since `has_sem` is a function, the proofs of the translated theorems are computationally reflexive again.

Conclusion of parts 3 and 4. When a theorem is being proved in HOL Light, its proof is recorded. It can be exported as an object `p: proof` that we expect to be correct (because it comes from a HOL Light proof). Given this object, we define a set of hypotheses `h` and a conclusion `t` with `proof2deriv`. `proof2deriv_correct` gives a computationally reflexive proof that these objects correspond to a derivation. If `h` is empty, we can apply `has_sem` to `t`, to get the Coq version of this theorem (which is very close to one would have written directly in Coq), and we have a computationally reflexive proof of this theorem applying `sem_deriv`.

5 Improving Efficiency

As such this process allows, in principle, to import HOL Light theorems and to check them... In practice, it would take too much time and use too much memory. We stopped when the exported files reached 200 Gb; Obua reports a similar experience [19].

5.1 Sharing

Proofs. Obua uses the natural sharing for proofs that comes from the user: when a piece of proof is used more than twice, it is shared. This sharing is not optimal, and it depends on the user's implementation, but it is very simple, it does not need too much memory (using a hash-table to perform hash-consing on proofs would really increase the memory consumption), and it is sufficient to considerably reduce the size of the exported files.

In Coq, this sharing is kept by adding one constructor to the inductive type `proof`:

```
Inductive proof : Type :=
|...| Poracle : forall h t, deriv h t → proof.
```

For each `proof` coming from HOL Light, the corresponding derivation is immediately computed. It can be called in a following `proof` thanks to the constructor `Poracle`.

Types and terms. We share types and terms using standard hash-consing presented in [10].

5.2 Opaque Proofs

In Coq, even with sharing, objects of type `proof` can be arbitrary big. Our idea to avoid keeping them in memory is to:

- distribute the theorems coming from HOL Light into separate files, and when compiling the $(n+1)^{\text{th}}$ file, load only the statements of the theorems of the first n files, but not the opaque proofs (this can be done with the option `-dont-load-proofs` of Coq's compiler);
- put the objects of type `proof` inside Coq opaque proofs.

5.3 Computation

In addition to its internal reduction mechanism, Coq includes an optimized bytecode-based virtual machine to evaluate terms. It is less tunable, but rather more efficient that the internal mechanism. It is well suited for the full evaluation required by computational reflection.

6 Results

6.1 Implementation

Our implementation is free software and can be found online [1]. Looking back at the two main objectives of this work, efficiency and usability, we observe some limitations to the first goal, while the second one is rather fulfilled.

Table 1. Benchmarking the standard library and Model

Bench.	Number		Time			Memory		
	Theorems	Lemmas	Rec.	Exp.	Comp.	H.D.D.	Virt. OCaml	Virt. Coq
Stdlib	1,726	195,317	2 min 30	6 min 30	10h	218 Mb	1.8 Gb	4.5 Gb
Model	2,121	322,428	6 min 30	29 min	44h	372 Mb	5.0 Gb	7.6 Gb
Vectors	2,606	338,087	6 min 30	21 min	39h	329 Mb	3.0 Gb	7.5 Gb

6.2 Tests

The tests were realized on a virtual machine that is installed on a DELL server PowerEdge 2950, with 2 processors Intel Xeon E5405 (quad-core, 2GHz) and 8 Gb RAM, with CentOS (64 bits). We are able to import and to check in Coq HOL Light proofs from:

- The standard library: what is loaded by default when launching HOL Light.
- The Model directory: a proof of consistency and soundness of HOL Light in HOL Light [16] (which, again, enforces the confidence in HOL Light).
- The elementary linear algebra tools developed in `Multivariate/vectors.ml`.

The results are summed up in **Table 1**. For each benchmark, we report the number of theorems that were exported, the number of lemmas generated by sharing, the time to interpret theorems and record their proofs in HOL Light, the time to export theorems to Coq, the time of compilation in Coq, the size of the generated Coq files, the maximal virtual memory used by OCaml, and the maximal virtual memory used by Coq. In the next two paragraphs, we analyze the origins of compilation time and memory consumption, and present some possible solutions.

Time and memory in Coq. Our proof sharing system has the major drawback to lead to a complete blow-up of the number of exported statements, as the first two columns of **Table 1** attest. Moreover, all these statements need to be kept in memory because all the theorems depend on one another.

The time of Coq's compilation thus cannot be less than quadratic in the number of Coq files, since compiling the $(n+1)^{\text{th}}$ file imports files 1 to n. The other operations that are expensive in time are:

- the parsing of the proof objects;
- the evaluation of the computationally reflexive proofs.

Concerning this last item, it is important to notice that Coq's virtual machine can run such a big computation. Computational reflection sounds thus a good way to import proofs, at least in Coq.

In other words: sharing limits the memory consumed by *proof objects*, but the resulting number of *statements* then becomes a problem. The compilation time is not too restrictive, since the incoming theorems have to be compiled once

Table 2. Mapping of definitions. In Coq, `N` is the type of binary natural numbers, and is defined in `NArith`. In HOL Light, `num` is the type of unary natural numbers.

HOL Light	num	+	*	DIV	MOD
Coq	N	Nplus	Nmult	Ndiv	Nmod

for all. Moreover, it requires far less human work to automatically export some theorems and compile it with our tool than to prove them from scratch in Coq. Memory is a large limitation for a user though, since he or she needs to import all the Coq files even to use only the last theorem. It would be convenient to be able not to load the intermediary lemmas, but it does not seem possible with our present proof objects implementation.

Memory in OCaml. The fact that proofs are kept is not the only limiting factor for OCaml's memory: during exportation, we create big hash-tables to perform hash-consing and to remember theorem statements. If we keep the present proof format, we definitely would have to reduce the extra-objects we construct for exportation.

Conclusion. Now that we have something reliable for a rather simple proof format, a next step is to switch to a more compact format such as the ones presented in [8] or [17].

6.3 Usability

We now give an example from integer arithmetic of an interaction between HOL Light and Coq's theorems. We map HOL Light's definitions as stated in Table 2.

Given the usual notation to say "a divides b":

```
Notation "a|b" := (Nmod b a = 0).
```

we import the theorem `MOD_EQ_0` from HOL Light:

```
Theorem hollight_MOD_EQ_0_thm :
  forall x x0 : N, x0 <> 0 →
    x0 | x = (exists a : N, x = a * x0).
```

and combine it with one Coq's theorem using tactics to prove:

```
Lemma div_mult : forall a b, a <> 0 → a | b →
   forall k, a | k*b.
```

The proof is only five lines long, because it is straightforward from `hollight_MOD_EQ_0_thm`. As Coq's standard library does not have any lemmas about division and modulo in `N`, proving this statement from scratch would certainly not be trivial.

7 Conclusion and Future Work

The new way to translate theorems and proofs from HOL Light to Coq presented in this paper fills the gap between those two interactive theorem provers. We solve both theoretical and practical problems coming from the different frameworks the two provers are based on. Relying on the computational power of Coq (reflection), our translation is both able to restore theorems meanings in Coq and give a small proof of them.

The implementation scales up to non trivial libraries. We are limited by usual performance issues, but at a much later point. To manage to import even larger developments, like the geometric properties used in the Flyspeck project [3], we need to reduce compilation time and virtual memory used. This may be possible by improving the proof format and changing the way we perform sharing.

Possible future directions include:

- translating HOL proofs to other systems with rich computation capabilities;
- integrating other external tools in Coq, such as other interactive theorem provers or automatic theorem provers, without compromising its soundness.

Acknowledgments. We are particularly grateful to Carlos Simpson who suggested the use of normalization by evaluation for translating HOL proofs to Coq and provided the stimulus for this work. We also thank John Harrison and Steven Obua who provided useful help with the code of HOL Light. We finally thank the anonymous reviewers for their useful remarks.

References

1. Our implementation, http://perso.ens-lyon.fr/chantal.keller/Recherche/hollightcoq.html
2. Dedukti, a universal proof checker, http://www.lix.polytechnique.fr/dedukti
3. The Flyspeck project, http://www.flyspeck-blog.blogspot.com
4. Aagaard, M., Harrison, J. (eds.): TPHOLs 2000. LNCS, vol. 1869. Springer, Heidelberg (2000)
5. Aydemir, B., Charguéraud, A., Pierce, B., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G., Wadler, P. (eds.) POPL, pp. 3–15. ACM, New York (2008)
6. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant: reference manual. Rapport technique - INRIA (2000)
7. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda-calculus. In: LICS, pp. 203–211. IEEE Computer Society, Los Alamitos (1991)
8. Berghofer, S., Nipkow, T.: Proof terms for simply typed higher order logic. In: Aagaard, Harrison: [4], pp. 38–52
9. Denney, E.: A prototype proof translator from hol to coq. In: Aagaard, Harrison: [4], pp. 108–125

10. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Kennedy, A., Pottier, F. (eds.) ML, pp. 12–19. ACM, New York (2006)
11. Furbach, U., Shankar, N. (eds.): IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
12. Garillot, F., Werner, B.: Simple types in type theory: Deep and shallow encodings. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 368–382. Springer, Heidelberg (2007)
13. Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq system. Tech. rep., INRIA (2007)
14. Gordon, M., Melham, T.: Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, New York (1993)
15. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
16. Harrison, J.: Towards self-verification of HOL Light. In: Furbach, Shankar: [11], pp. 177–191
17. Hurd, J.: OpenTheory: Package Management for Higher Order Logic Theories. In: PLMMS'09, p. 31 (2009)
18. Leroy, X.: The OCaml Programming Language (1998)
19. Obua, S., Skalberg, S.: Importing HOL into Isabelle/HOL. In: Furbach, Shankar: [11], pp. 298–302
20. Wiedijk, F.: Encoding the HOL Light logic in Coq. Unpublished notes (2007)
21. Wong, W.: Recording and checking HOL proofs. In: Schubert, E., Windley, P., Alves-Foss, J. (eds.) TPHOLs 1995. LNCS, vol. 971, pp. 353–368. Springer, Heidelberg (1995)
22. Zammit, V.: A comparative study of Coq and HOL. In: Gunter, E., Felty, A. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 323–337. Springer, Heidelberg (1997)

A Mechanized Translation from Higher-Order Logic to Set Theory

Alexander Krauss and Andreas Schropp

Technische Universität München, Institut für Informatik
{krauss,schropp}@in.tum.de

Abstract. In order to make existing formalizations available for set-theoretic developments, we present an automated translation of theories from Isabelle/HOL to Isabelle/ZF. This covers all fundamental primitives, particularly type classes. The translation produces LCF-style theorems that are checked by Isabelle's inference kernel. Type checking is replaced by explicit reasoning about set membership.

1 Introduction

Compared to the type theories underlying most widely-known proof assistants today, set theory has received less attention in the field of interactive theorem proving. This is unfortunate, since set theory is arguably the formalism that comes closest to a "standard foundation of mathematics" and since it provides a rich and well-understood foundation.

This paper describes an automated translation of theories from Isabelle/HOL to Isabelle/ZF (which implements ZFC). We interpret recorded proof terms, and the resulting derivations are again checked by Isabelle's LCF-style inference kernel, which ensures soundness of the approach and implementation.

While the general idea of a mapping to set theory is not new—in fact, the standard semantics of HOL [17] is defined in ZFC—, translating entire theories of realistic proof developments is a highly non-trivial task: In addition to the bare proofs, one must cope with theory extension mechanisms like constant and type definitions. Moreover, Isabelle's type classes and overloading as well as interactions between the object-logic Isabelle/HOL and the logical framework Isabelle/Pure complicate this task. To our knowledge, this is the first complete translation scheme to set theory (it is complete except for a fine point, discussed in §5.2). It is also the first proof-producing one.

1.1 Motivation

The motivation for this work comes from several directions:

Experimenting with theorem proving based on set theory. The simple type theory of HOL is sometimes a limitation that makes certain concepts (e.g., monads, which would require parametrization over type constructors) hard or impossible

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 323–338, 2010.

to formalize. One way of improving this is to move to a stronger type theory, like the calculus of constructions [2] or the recently proposed HOL Omega [8]. An alternative path is to abandon types as an integral part of the logic and to work in a logic that is untyped, but expresses the equivalent of typing judgements explicitly as propositions. A type discipline could then be reintroduced as an extra layer of "soft types" built on top of an untyped LCF kernel. Such a system could make the notion of type checking more open to experimentation, since it is easier to change something that is not part of the foundation.

This idea has been mentioned in the literature several times [5,9,20,6], and, in principle, Isabelle/ZF could be a starting point to explore these possibilities. This work intends to explore how HOL-style reasoning can be turned into set-theoretic reasoning, mechanized in Isabelle/ZF.

Exchange format between proof assistants. Although there already exist translation facilities for proofs between different theorem provers, combining developments from different systems in practice is still an open problem (Gordon calls it a "Grand Challenge" [6]). Set theory has sometimes been mentioned as a candidate for an exchange format between different logics, mainly because the semantics of many logical systems can be defined set-theoretically.

Reviving Isabelle/ZF. Our translation makes the large body of theories developed in Isabelle/HOL available in ZF. We believe that this may facilitate the development of proof tools (e.g., arithmetic decision procedures) in set theory, which typically require a certain amount of established theory.

None of the goals that we take as a long-term motivation can be solved by this work alone. However, we aim to take a small step towards them with the following concrete contributions:

- By carefully revisiting the foundations of the Isabelle/HOL system, we show how all its primitives can be translated to a purely definitional theory in Isabelle/ZF (with global choice; see §3). In particular, type classes and overloading are eliminated.
- We provide a prototype implementation of this mapping that produces machine-checked proofs in Isabelle/ZF.

1.2 Related Work

The standard set-theoretic semantics of HOL is described by Pitts [17]. Gordon [5] experimented with combinations of HOL and ZFC by axiomatizing a type of sets in HOL. He describes a transfer principle between the two worlds which is very similar to our translation of propositions. However, Gordon's translation is not proof-producing, and one must trust the correctness of its implementation. Moreover, the semantics of the axiomatic combination of HOL and ZFC are still slightly unclear.

A number of proof transformation tools have been developed to replay proofs of one theorem prover in another, mostly within the HOL family of provers [10,14]. Similarly, the AWE extension [1] interprets theories within the

Isabelle/HOL system, replacing types, constants, and axioms with concrete models. Our translation is closely related to these tools but slightly more complex, since our target language is untyped and type reasoning has to be made explicit.

2 Formal Preliminaries

Isabelle [15] is a generic theorem prover, which supports reasoning in different object-logics embedded in a logical framework (often referred to as the "meta-logic"). While many Isabelle applications use Isabelle/HOL exclusively, this particular work critically relies on the generic nature of the system.

In this section, we recall the logical foundations of Isabelle, including how the object-logics HOL and ZF are embedded in Isabelle/Pure.

The meta-syntactic abbreviation $\overline{t_m}$ always denotes the sequence $t_1 \dots t_m$. Binding a sequence of variables $(\lambda\, \overline{x_m : \tau_m}.\ t)$ always means iteration of binders $(\overline{\lambda\, x_m : \tau_m}.\ t)$. We omit type and sort annotations when they are clear from the context. Substitution of t_2 for x in t_1 is written as $t_1[x := t_2]$.

2.1 Pure

Isabelle/Pure is a simply-typed intuitionistic higher-order logic featuring just implication, universal quantification, equality and schematic polymorphism with type classes [19,7]. Unlike many dependently-typed systems, it retains the stratification into sorts, types, terms, and proofs, which we discuss in this order.

Sorts and Types. Syntactically, (type) classes c are formal names, and sorts s are finite symbolic intersections of classes, written as finite sets $\{c_1, \dots, c_n\}$, where the empty intersection is denoted by $\top$. Types are either type constructor applications, or type variables annotated with their sort.

$$\tau \quad ::= \quad \kappa\ \overline{\tau_m} \quad \mid \quad \alpha^s$$

Special type constructors are *prop* (propositions) and $\Rightarrow$ (function space).

A set of type classes together with an acyclic subclass relation $\prec$ and a set A of arities of the form $\kappa :: (\overline{s_m})c$ is called an order-sorted algebra [18]. It induces the type-in-class relation $\tau : c$ defined by the following rules, where $\tau : \{c_1, \dots, c_n\}$ abbreviates $\tau : c_1, \dots, \tau : c_n$.

$$\frac{\tau : c_1 \qquad c_1 \prec c_2}{\tau : c_2} \qquad \frac{\overline{\tau_m : s_m} \qquad (\kappa :: (\overline{s_m})c) \in A}{\kappa\ \overline{\tau_m} : c} \qquad \frac{c \in s}{\alpha^s : c}$$

For now, we regard classes and sorts as purely syntactic. The details, including the interaction of type classes and derivations, are deferred to §5.

Terms. The language of terms is a conventional simply-typed lambda calculus extended with constants (also denoted by c), whose types can be instantiated

at each occurrence. This yields schematic polymorphism, where type inference is still decidable, since arbitrary type abstractions are not allowed.

$$t \quad ::= \quad x \quad | \quad c[\overline{\tau_m}] \quad | \quad t_1\ t_2 \quad | \quad \lambda x : \tau.\ t$$

We also write c for $c[]$. Primitive constants include universal quantification $\bigwedge : (\alpha^\top \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop}$, implication $\Longrightarrow : \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$, and equality $\equiv : \alpha^\top \Rightarrow \alpha^\top \Rightarrow \mathit{prop}$. A term is called *closed* if it contains no free term variables.

The typing rules for terms are standard. We assume a function Σ that maps constants to their types with canonical type variables $\overline{\alpha_m^\top}$.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash c[\overline{\tau_m}] : \Sigma(c)[\overline{\alpha_m^\top := \tau_m}]}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ t) : \tau_1 \Rightarrow \tau_2}$$

Terms of type *prop* are called propositions and are denoted by ϕ.

Proofs. The language of proofs constitutes another level of lambda calculus on top of terms, in the spirit of the Curry-Howard correspondence. The two versions of abstraction and application correspond to the introduction and elimination of $\bigwedge$ and $\Longrightarrow$, respectively. Proof variables h stand for hypotheses. Axioms and previously-proved theorems are modelled as proof constants *thm*, whose types can be instantiated in a manner similar to term constants.

$$p \quad ::= \quad \lambda x : \tau.\ p \quad | \quad \lambda h : \phi.\ p \quad | \quad p \odot t \quad | \quad p_1 \bullet p_2 \quad | \quad h \quad | \quad \mathit{thm}[\overline{\tau_m}]$$

Note that *thm* is a meta variable for proof constants. We also write *thm* for *thm*[].

We give the main propositions-as-types typing rules for proofs, which are to be read modulo $\alpha\beta\eta$-equivalence. Like for term constants, the function Σ yields the proposition proved by a proof constant, which may contain free type variables $\overline{\alpha_m^{s_m}}$.

$$\frac{\Gamma, x : \tau \vdash p : \phi}{\Gamma \vdash (\lambda x : \tau.\ p) : (\textstyle\bigwedge x : \tau.\ \phi)} \qquad \frac{\Gamma,\ h : \phi_1 \vdash p : \phi_2}{\Gamma \vdash (\lambda h : \phi_1.\ p) : \phi_1 \Longrightarrow \phi_2}$$

$$\frac{\Gamma \vdash p : (\textstyle\bigwedge x : \tau.\ \phi) \qquad \Gamma \vdash t : \tau}{\Gamma \vdash (p \odot t) : \phi[x := t]} \qquad \frac{\Gamma \vdash p_1 : \phi_1 \Longrightarrow \phi_2 \qquad \Gamma \vdash p_2 : \phi_1}{\Gamma \vdash (p_1 \bullet p_2) : \phi_2}$$

$$\frac{h : \phi \in \Gamma}{\Gamma \vdash h : \phi} \qquad \frac{\overline{\tau_m : s_m}}{\Gamma \vdash \mathit{thm}[\overline{\tau_m}] : \Sigma(\mathit{thm})[\overline{\alpha_m^{s_m} := \tau_m}]}$$

Pure has further rules and axioms, ensuring that $\equiv$ is an extensional and congruent equivalence relation, which equates $\alpha\beta\eta$-convertible lambda terms and is the bi-implication on propositions.

Sometimes it is convenient to imagine that all type variables $\overline{\alpha_m^{s_m}}$ occurring in a closed proposition ϕ are explicitly quantified on the outermost level. We take the freedom to write this as $\forall \overline{\alpha_m : s_m}.\ \phi$. Similarly, on the level of proofs, the dependency on type variables is expressed using an abstraction-like notation $\lambda\, \overline{\alpha_m : s_m}.\ p$. A proof is called *closed* if all occuring term and type variables are bound in this way. However, these notations are not part of the formal system!

Constant definitions. Pure allows constant definitions, i.e., the introduction of a new constant $c : \tau$ and the definitional axiom $c[\overline{\alpha_m}] \equiv t$, if τ contains exactly the type variables $\overline{\alpha_m}$, t is closed, $t : \tau$, and t contains only type variables $\overline{\alpha_m}$. Since defining equations can always be unfolded, this is a conservative theory extension. Defining equations are written with $:\equiv$ instead of $\equiv$.

2.2 HOL

Higher-order logic (HOL) is embedded as an object-logic in Isabelle/Pure by introducing a type *bool* of classical truth values. A constant $\mathit{Trueprop} : \mathit{bool} \Rightarrow \mathit{prop}$ embeds booleans into propositions. We write $\mathit{Trueprop}\ t$ as $[\,t\,]$. Object-level quantifiers and connectives are introduced as constants $\forall : (\alpha \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$; $\neg : \mathit{bool} \Rightarrow \mathit{bool}$; $\longrightarrow, \vee, \wedge : \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$; $= : \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$, etc.

Natural-deduction rules $\dfrac{A \qquad B}{C}$ can then be expressed as meta-level propositions $A \Longrightarrow B \Longrightarrow C$. For example, these are the introduction and elimination rules for $\forall$ and $\longrightarrow$, and the law of the excluded middle:

$$\begin{aligned}
\text{allI}: &\quad \bigwedge P : \alpha \Rightarrow \mathit{bool}.\ (\bigwedge x : \alpha.\ [\,P\ x\,]) \Longrightarrow [\,\forall x.\ P\ x\,] \\
\text{spec}: &\quad \bigwedge (P : \alpha \Rightarrow \mathit{bool})\ (a : \alpha).\ [\,\forall x.\ P\ x\,] \Longrightarrow [\,P\ a\,] \\
\text{impI}: &\quad \bigwedge P\, Q : \mathit{bool}.\ ([\,P\,] \Longrightarrow [\,Q\,]) \Longrightarrow [\,P \longrightarrow Q\,] \\
\text{mp}: &\quad \bigwedge P\, Q : \mathit{bool}.\ [\,P \longrightarrow Q\,] \Longrightarrow [\,P\,] \Longrightarrow [\,Q\,] \\
\text{True_or_False}: &\quad \bigwedge P : \mathit{bool}.\ [\,P = \mathit{True} \vee P = \mathit{False}\,]
\end{aligned}$$

In Isabelle, outermost quantifiers and the $[\,\cdot\,]$-embedding are not printed, such that the first rule reads $(\bigwedge x : \alpha.\ P\ x) \Longrightarrow \forall x.\ P\ x$, but we will keep them explicit in this paper, to visualize the division between meta- and object-logic.

HOL has some more rules and axioms, which we omit for brevity. However, two primitive constants are notable: the definite description operator $\mathit{THE} : (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha$, axiomatized by $\bigwedge a.[\,(\mathit{THE}\ x.\ x = a) = a\,]$, and the constant $\mathit{undefined} : \alpha$, which is unspecified as it comes with no axiom.

The approach of modelling the inference rules of the object-logic as theorems in the meta-logic is common to all of Isabelle's object logics. What is special in HOL is that the function space of the object-logic coincides with that of the meta-logic. This works because HOL and Pure are so similar. To make interactions with the framework more explicit, a type class *hol* is used to characterize HOL types.

This class contains types such as *bool*, natural numbers, lists, and it is closed under $\Rightarrow$. Types such as *prop* or $\mathit{bool} \Rightarrow \mathit{prop}$ are not in this class. The HOL axioms are restricted to types which are in the *hol* class.

Type definitions. A speciality of HOL is the ability to define new type constructors from non-empty subsets of existing types. A type definition $\kappa\ \overline{\alpha_m} \cong P$, where $P : \tau \Rightarrow \mathit{bool}$ is a closed term (called the *representing set*) with type variables $\overline{\alpha_m}$, together with a proof of $[\,\exists x.\ P\ x\,]$, gives rise to a bijection between $\kappa\ \overline{\alpha_m}$ and τ, in the form of new constants $\mathit{Rep}_\kappa : \kappa\ \overline{\alpha_m} \Rightarrow \tau$, $\mathit{Abs}_\kappa : \tau \Rightarrow \kappa\ \overline{\alpha_m}$ and an axiom

$$\begin{aligned}[\ & (\forall z : \kappa\ \overline{\alpha_m}.\ P\ (\mathit{Rep}_\kappa[\overline{\alpha_m}]\ z)) \wedge \\ & (\forall z : \kappa\ \overline{\alpha_m}.\ \mathit{Abs}_\kappa[\overline{\alpha_m}]\ (\mathit{Rep}_\kappa[\overline{\alpha_m}]\ z) = z) \wedge \\ & (\forall y : \tau.\ P\ y \longrightarrow \mathit{Rep}_\kappa[\overline{\alpha_m}]\ (\mathit{Abs}_\kappa[\overline{\alpha_m}]\ y) = y)\ ],\end{aligned}$$

which we abbreviate as $\mathrm{typedef}_\kappa$. Notice that this axiom does not specify the value of $\mathit{Abs}_\kappa[\overline{\alpha_m}]\ y$ when $[\,P\ y\,]$ does not hold.

2.3 ZF

First-order logic (FOL), the basis of ZFC, is modelled in Isabelle by two types ι and o, representing individuals and truth values, respectively. Again, an embedding $[\,\cdot\,] : o \Rightarrow \mathit{prop}$ is introduced, along with first-order connectives $\forall : (\iota \Rightarrow o) \Rightarrow o$; $\neg : o \Rightarrow o$; $\longrightarrow, \vee, \wedge : o \Rightarrow o \Rightarrow o$; $= : \iota \Rightarrow \iota \Rightarrow o$, etc., and the usual natural deduction rules for them. ZFC is then added using the standard collection of FOL axioms (see [16] for details) about the constant $\in : \iota \Rightarrow \iota \Rightarrow o$. Note that the axiom schema of replacement is represented as Pure quantification over a predicate. As ι is the type of all sets, the meta level can be used to reason about proper classes ($\iota \Rightarrow o$), operators ($\iota \Rightarrow \iota$), binding structures ($(\iota \Rightarrow \tau) \Rightarrow \tau$), etc.

Function spaces can be constructed using $\to : \iota \Rightarrow \iota \Rightarrow \iota$, and elements of $A \to B$ can be constructed using an operator $\mathit{Lambda} : \iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota$, which restricts the domain of an operator $f : \iota \Rightarrow \iota$ to a set A. We write $(\lambda x \in A.\ f\ x)$ for $\mathit{Lambda}\ A\ f$. The application of such a function to an argument is written using an explicit apply operator $\text{`} : \iota \Rightarrow \iota \Rightarrow \iota$. While α-conversion is inherited from the framework, β- and η-reduction are conditional rewrite rules.

$$\begin{aligned}\textstyle\bigwedge A\ y\ f.\ [\,y \in A\,] &\Longrightarrow (\lambda x \in A.\ f\ x)\text{`}y \equiv f\ y \\ \textstyle\bigwedge A\ B\ f.\ [\,f \in A \to B\,] &\Longrightarrow (\lambda x \in A.\ f\text{`}x) \equiv f\end{aligned} \qquad (\mathsf{ZF}\text{-}\beta\eta)$$

As opposed to HOL, where formulas are just special terms, in FOL the languages of formulas (o) and terms (ι) are syntactically separated. For the sake of uniformity, our translation will map everything to type ι, using the set $\mathbb{B} :\equiv \{0, 1\}$ for truth values, with the interpretation function $\langle\,\cdot\,\rangle :\equiv (\lambda x.\ x = 1) : \iota \Rightarrow o$. We must thus define appropriate versions of logical connectives, such as $\widehat{\mathit{True}}$, $\widehat{\mathit{False}}$, $\widehat{\longrightarrow} : \iota$ and $\mathbin{\widehat{=}}, \widehat{\forall} : \iota \Rightarrow \iota$.

$$\begin{aligned}\widehat{\mathit{True}} &:\equiv 1, \quad \widehat{\mathit{False}} :\equiv 0 \\ \widehat{\longrightarrow} &:\equiv (\lambda A, B \in \mathbb{B}.\ \text{if } A = \widehat{\mathit{False}} \text{ then } \widehat{\mathit{True}} \text{ else } B), \\ \mathbin{\widehat{=}} &:\equiv (\lambda A : \iota.\ \lambda x\ y \in A.\ \text{if } x = y \text{ then } \widehat{\mathit{True}} \text{ else } \widehat{\mathit{False}}), \\ \widehat{\forall} &:\equiv (\lambda A : \iota.\ \lambda P \in A \to \mathbb{B}.\ P =_{A \to \mathbb{B}} (\lambda x \in A.\ \widehat{\mathit{True}}))\end{aligned}$$

$(\mathbin{\widehat{=}})A\ x\ y$ is written $x =_A y$, and $(\widehat{\forall}\,A\ (\lambda x \in A.\ P\ x))$ is written $(\widehat{\forall} x \in A.\ P\ x)$.

3 The Basic Translation

The standard set-theoretic model of HOL [17] is based on a set $\mathcal{U}$, which serves as the universe of types. Among other requirements, $\mathcal{U}$ must be closed under function spaces. For example, the set $V_{\omega+\omega} \setminus \{\emptyset\}$, with $V_{\omega+\omega}$ of the cumulative hierarchy (see, e.g., [11]), could be used as the set of all types. While such a relatively small model may be desirable from a foundational point of view, it would make the results of our translation weaker than necessary, and not very intuitive. In fact, it is not necessary that the universe of types forms a set, and so we prefer to use the proper class of all non-empty sets.

The idea underlying the translation is as follows.

- Types τ are mapped to terms denoting non-empty sets $[\![\tau]\!] : \iota$.
- Type constructors are mapped to operations on sets.
- Terms $t : \tau$ are mapped to terms $[\![t]\!] : \iota$, such that $[\![t]\!] \in [\![\tau]\!]$ holds.
- Application and abstraction are translated to $(\lambda x \in A.\ \ldots)$ and ‘.
- Quantification over types (which may only occur on the outermost level) is mapped to Pure quantification over sets.
- Type annotations on variables are mapped to set membership assumptions.
- Proofs are instrumented with non-emptiness and membership derivations, following the typing rules.

Example 1. The statement $\forall\alpha : \{hol\}.\ \bigwedge(x : \alpha)\,(P : \alpha \Rightarrow bool).\ [P\ x]$ is translated to

$$\bigwedge A : \iota.\ [A \neq \emptyset] \Longrightarrow (\bigwedge x : \iota.\ [x \in A] \Longrightarrow (\bigwedge P : \iota.\ [P \in A \to \mathbb{B}] \Longrightarrow [\langle P\,`\,x \rangle]))\,,$$

which, after moving quantifiers out, becomes

$$\bigwedge A\,x\,P : \iota.\ [A \neq \emptyset] \Longrightarrow [x \in A] \Longrightarrow [P \in A \to \mathbb{B}] \Longrightarrow [\langle P\,`\,x \rangle]\ .$$

Example 2. The transitivity rule for HOL equality,

$$\forall\alpha : \{hol\}.\ \bigwedge r\,s\,t : \alpha.\ [r = s] \Longrightarrow [s = t] \Longrightarrow [r = t]$$

is translated to

$$\begin{aligned}\bigwedge A : \iota.\ [A \neq \emptyset] \Longrightarrow (\bigwedge r : \iota.\ [r \in A] \Longrightarrow (\bigwedge s : \iota.\ [s \in A] \Longrightarrow \\ (\bigwedge t : \iota.\ [t \in A] \Longrightarrow [\langle r =_A s \rangle] \Longrightarrow [\langle s =_A t \rangle] \Longrightarrow [\langle r =_A t \rangle]))))\,,\end{aligned}$$

which, after moving quantifiers out, becomes

$$\begin{aligned}\bigwedge A\,r\,s\,t : \iota.\ [A \neq \emptyset] \Longrightarrow [r \in A] \Longrightarrow [s \in A] \Longrightarrow [t \in A] \\ \Longrightarrow [\langle r =_A s \rangle] \Longrightarrow [\langle s =_A t \rangle] \Longrightarrow [\langle r =_A t \rangle]\ .\end{aligned}$$

One consequence of our choice of translation is that we need an axiom of global choice, which postulates an operation *choose* satisfying

$$\bigwedge A : \iota.\ [A \neq \emptyset] \Longrightarrow [\mathit{choose}\ A \in A]\,.$$

This is a conservative extension of ZFC [4]. The need for it arises not only from the fact that HOL includes a choice operator by itself, but already from the presence of underspecification. While the constant *undefined* has no particular properties in HOL, its translation to ZF must satisfy at least the formula $\bigwedge A : \iota.\ [\, A \neq \emptyset \,] \Longrightarrow [\, \widehat{\mathit{undefined}}\ A \in A \,]$, which arises from its type. This is exactly the global choice axiom.

In the following presentation of the basic translation scheme, we make a few simplifying assumptions: First, we assume that the theorems we translate do not mix HOL and Pure arbitrarily, but use essentially plain HOL reasoning, except for an outermost layer of $\bigwedge$ and $\Longrightarrow$. Theorems relevant in practice typically have this form (see §6.2 for exceptions). Second, all type variables must be of sort $\{\mathit{hol}\}$ (see §5 for the treatment of other classes). Third, we assume that there are no type or term variables in proofs that do not occur in the corresponding proposition (which is easy to achieve by substituting any ground type or term) and that proofs are closed.

Type constructors κ, term constants c, and proof constants *thm* occurring in proofs must already have translations $\widehat{\kappa}$, $\widehat{c}$ and $\widehat{\mathit{thm}}$. These are either set up manually, as it must be done for the primitives and axioms, or come from a recursive invocation of the translation scheme.

Types and terms are translated as follows:

Translation of types:

$$\begin{aligned} [\![\, \kappa\ \overline{\tau_m} \,]\!] &:= \widehat{\kappa}\ \overline{[\![\, \tau_m \,]\!]} \\ [\![\, \alpha \,]\!] &:= x_\alpha \end{aligned}$$

Translation of terms:

$$\begin{aligned} [\![\, c[\overline{\tau_m}] \,]\!] &:= \widehat{c}\ \overline{[\![\, \tau_m \,]\!]} \\ [\![\, \lambda x : \tau.\ t \,]\!] &:= \lambda x \in [\![\, \tau \,]\!].\ [\![\, t \,]\!] \\ [\![\, x \,]\!] &:= x \\ [\![\, t_1\ t_2 \,]\!] &:= [\![\, t_1 \,]\!]\ `\ [\![\, t_2 \,]\!] \end{aligned}$$

Note that for a type τ, $[\![\, \tau \,]\!]$ is a term (of type ι), not a type. Type instantiations of constants are translated to applications.

In the outer structure of propositions, the domain of universal quantifiers is restricted to the respective sets, and the *Trueprop* embedding is replaced with $[\, \langle \cdot \rangle \,]$. On the outermost level, non-emptiness conditions are added for the sets arising from type variables.

Translation of outer proposition structure:

$$\begin{aligned} [\![\, \textstyle\bigwedge x : \tau.\ \phi \,]\!] &:= \textstyle\bigwedge x : \iota.\ [\, x \in [\![\, \tau \,]\!] \,] \Longrightarrow [\![\, \phi \,]\!] \\ [\![\, \phi_1 \Longrightarrow \phi_2 \,]\!] &:= [\![\, \phi_1 \,]\!] \Longrightarrow [\![\, \phi_2 \,]\!] \\ [\![\, [\, t \,] \,]\!] &:= [\, \langle\, [\![\, t \,]\!] \,\rangle \,] \\ [\![\, \forall \overline{\alpha_m} : \{\mathit{hol}\}.\ \phi \,]\!] &:= \textstyle\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\![\, \phi \,]\!] \end{aligned}$$

In the proof transformation given below, the proofs corresponding to typing judgements and non-emptiness of types must be filled in explicitly. We mark the positions where a proof of $[\, P \,]$ must be inserted by placeholders $\{P\}$. This proof, which may refer to hypotheses available in the respective context, is generated by

means of a tactic. Moreover, since the original proof is modulo $\alpha\beta\eta$-equivalence and abstraction and application have been translated to their ZF counterparts, we must explicitly normalize them by rewriting with the rules (ZF-$\beta\eta$). For a proposition ϕ, let $\mathsf{norm}\,(\phi)$ denote the normalized proposition. For a proof $p : \phi$, $\mathsf{norm}\,(p)$ denotes a proof of $\mathsf{norm}\,(\phi)$, and $p_1 \bullet_{\mathsf{n}} p_2$ abbreviates $\mathsf{norm}\,(p_1) \bullet \mathsf{norm}\,(p_2)$. We generate the proof of $\mathsf{norm}\,(p)$ from p using Isabelle's simplifier.

Translation of proofs:

$$
\begin{array}{rcl}
[\![\lambda x : \tau.\ p]\!] & := & \lambda x : \iota.\ \lambda h : [\, x \in [\![\tau]\!] \,].\ [\![p]\!] \\
[\![\lambda h : \phi.\ p]\!] & := & \lambda h : [\![\phi]\!].\ [\![p]\!] \\
[\![h]\!] & := & h \\
[\![p_1 \bullet p_2]\!] & := & [\![p_1]\!] \bullet_{\mathsf{n}} [\![p_2]\!] \\
[\![p \odot t]\!] & := & [\![p]\!] \odot [\![t]\!] \bullet_{\mathsf{n}} \{ [\![t]\!] \in [\![\tau]\!] \} \quad \text{where } t : \tau \\
[\![\lambda \overline{\alpha_m} : \{hol\}.\ p]\!] & := & \lambda \overline{x_{\alpha_m}} : \iota.\ \lambda \overline{h_m : [\, x_{\alpha_m} \neq \emptyset \,]}.\ [\![p]\!] \\
[\![thm[\overline{\tau_m}]]\!] & := & \widehat{thm} \odot \overline{[\![\tau_m]\!]} \bullet_{\mathsf{n}} \overline{\{ [\![\tau_m]\!] \neq \emptyset \}}
\end{array}
$$

Example 3. The proof of the transitivity rule shown previously is based on a substitution rule, one of HOL's axioms:

$$
\begin{array}{l}
\mathrm{subst} : (\forall \alpha : \{hol\}.\ \bigwedge (s\, t : \alpha)\, (P : \alpha \Rightarrow bool).\ [\, s = t \,] \Longrightarrow [\, P\ s \,] \Longrightarrow [\, P\ t \,]) \\
\widehat{\mathrm{subst}} : (\bigwedge x_\alpha : \iota.\ [\, x_\alpha \neq \emptyset \,] \Longrightarrow (\bigwedge s : \iota.\ [\, s \in x_\alpha \,] \Longrightarrow (\bigwedge t : \iota.\ [\, t \in x_\alpha \,] \Longrightarrow \\
\qquad (\bigwedge P : \iota.\ [\, P \in x_\alpha \to \mathbb{B} \,] \Longrightarrow [\langle\, s =_{x_\alpha} t \,\rangle] \Longrightarrow [\langle\, P\, `\, s \,\rangle] \Longrightarrow [\langle\, P\, `\, t \,\rangle]))))) \\
\mathrm{trans} = (\lambda\, (\alpha : \{hol\})\, (r\, s\, t : \alpha)\, (h : [\, r = s \,])\, (h' : [\, s = t \,]). \\
\qquad \mathrm{subst}[\alpha] \odot s \odot t \odot (\lambda x : \alpha.\ r = x) \bullet h' \bullet h) \\
\qquad : (\forall \alpha : \{hol\}.\ \bigwedge (r\, s\, t : \alpha).\ [\, r = s \,] \Longrightarrow [\, s = t \,] \Longrightarrow [\, r = t \,]) \\
\widehat{\mathrm{trans}} = (\lambda\, (x_\alpha : \iota)\, (h_\alpha : [\, x_\alpha \neq \emptyset \,])\, (r : \iota)\, (h_r : [\, r \in x_\alpha \,])\, (s : \iota)\, (h_s : [\, s \in x_\alpha \,]). \\
\qquad (\lambda\, (t : \iota)\, (h_t : [\, t \in x_\alpha \,])\, (h : [\langle\, r =_{x_\alpha} s \,\rangle])\, (h' : [\langle\, s =_{x_\alpha} t \,\rangle]). \\
\qquad\quad \widehat{\mathrm{subst}} \odot x_\alpha \bullet_{\mathsf{n}} \{ x_\alpha \neq \emptyset \} \odot s \bullet_{\mathsf{n}} \{ s \in x_\alpha \} \odot t \bullet_{\mathsf{n}} \{ t \in x_\alpha \} \\
\qquad\qquad \odot (\lambda x \in x_\alpha.\ r =_{x_\alpha} x) \bullet_{\mathsf{n}} \{ (\lambda x \in x_\alpha.\ r =_{x_\alpha} x) \in x_\alpha \to \mathbb{B} \} \\
\qquad\qquad \bullet_{\mathsf{n}} h' \bullet_{\mathsf{n}} h))
\end{array}
$$

4 Translating Constant and Type Definitions

When translating constant definitions $c[\overline{\alpha_m}] :\equiv t$ where $t : \tau$, the dependency on types $\overline{\alpha_m}$ is made explicit. We introduce a new constant $\widehat{c} : \iota^m \Rightarrow \iota$.

$$\widehat{c} :\equiv (\lambda \overline{x_{\alpha_m}} : \iota.\ [\![t]\!])$$

The translation of the original definition $[\![c[\overline{\alpha_m}] \equiv t]\!]$ (that is, $\widehat{c}\ \overline{x_{\alpha_m}} \equiv [\![t]\!]$) is a simple consequence of the above definition and proved automatically. Moreover, the following theorem is deduced, which is needed by the type checking tactic to derive the translation of typing judgements involving c.

$$\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\, x_{\alpha_m} \neq \emptyset \,]} \Longrightarrow [\ \widehat{c}\ \overline{x_{\alpha_m}} \in [\![\tau]\!] \,]$$

Type definitions $\kappa\ \overline{\alpha_m} \cong P$ with $P : \tau \Rightarrow \mathit{bool}$ are translated to constant definitions of a set $\widehat{\kappa}$ (parameterized by the sets $\overline{x_{\alpha_m}}$ arising from type parameters) and two functions $\widehat{\mathit{Rep}_\kappa}$ and $\widehat{\mathit{Abs}_\kappa}$.

$$\begin{aligned}
\widehat{\kappa} &:\equiv \lambda\, \overline{x_{\alpha_m}} : \iota.\ \{z \in [\![\tau]\!] \mid \langle [\![P]\!] \text{`} z \rangle\} \\
\widehat{\mathit{Rep}_\kappa} &:\equiv \lambda\, \overline{x_{\alpha_m}} : \iota.\ \lambda\, z \in \widehat{\kappa}\ \overline{x_{\alpha_m}}.\ z \\
\widehat{\mathit{Abs}_\kappa} &:\equiv \lambda\, \overline{x_{\alpha_m}} : \iota.\ \lambda\, y \in [\![\tau]\!].\ \text{if } y \in \widehat{\kappa}\ \overline{x_{\alpha_m}} \text{ then } y \text{ else } \widehat{\mathit{undefined}}\ (\widehat{\kappa}\ \overline{x_{\alpha_m}})
\end{aligned}$$

Since the new type is simply mapped to a subset of the original type, the functions $\widehat{\mathit{Rep}_\kappa}$ and $\widehat{\mathit{Abs}_\kappa}$ become identity mappings. Since $\widehat{\mathit{Abs}_\kappa}$ must be total and always return an element of $\widehat{\kappa}$ to satisfy its type, we use $\widehat{\mathit{undefined}}$.

From these definitions we derive the characteristic property $[\![\mathrm{typedef}_\kappa]\!]$ for the type definition, as well as the typing lemmas for $\widehat{\mathit{Rep}_\kappa}$ and $\widehat{\mathit{Abs}_\kappa}$ and the fact that the new type constructor preserves non-emptiness. The proof of the latter theorem makes use of the non-emptiness proof provided for the original HOL definition.

$$\begin{aligned}
&\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\,x_{\alpha_m} \neq \emptyset\,]} \Longrightarrow [\ \widehat{\mathit{Rep}_\kappa}\ \overline{x_{\alpha_m}} \in \widehat{\kappa}\ \overline{x_{\alpha_m}} \to [\![\tau]\!]\ ] \\
&\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\,x_{\alpha_m} \neq \emptyset\,]} \Longrightarrow [\ \widehat{\mathit{Abs}_\kappa}\ \overline{x_{\alpha_m}} \in [\![\tau]\!] \to \widehat{\kappa}\ \overline{x_{\alpha_m}}\ ] \\
&\bigwedge \overline{x_{\alpha_m}} : \iota.\ \overline{[\,x_{\alpha_m} \neq \emptyset\,]} \Longrightarrow [\ \widehat{\kappa}\ \overline{x_{\alpha_m}} \neq \emptyset\ ]
\end{aligned}$$

5 Translating Type Classes and Overloaded Definitions

The translation described so far covers standard HOL. However, in the Isabelle/HOL libraries, even the most basic theories make heavy use of type classes and overloading, which means that our translation must support them to be practically useful.

The basic solution is to employ a preprocessing step which eliminates classes and overloading from theories, producing a theory in plain HOL. Then, the translation from the previous section can be applied to obtain the set-theoretic version.

In this section, we describe the basics of type classes and overloading, which were neglected in §2. Then we show how to compile them away. Although the two mechanisms are typically used together (cf. [7]), we can treat them separately, removing first type classes and then overloading. The outline of this transformation was already pointed out by Haftmann and Wenzel [7], but not considering proof terms and without implementation.

5.1 Type Classes

In a nutshell, classes assigned to a type τ express extra properties of τ that are propagated by the type system. In particular, a sort annotation s on a type variable α^s corresponds to an implicit hypothesis about α.

It is possible to embed types into the term language using a unary type constructor *itself* and a constant $TYPE : itself\ \alpha$, such that the type τ can be represented by the term $TYPE[\tau]$.

To describe the logical properties of types in a class c, a constant $\mathrm{Cl}_c : itself\ \alpha \Rightarrow prop$ is defined. For example, the class of all finite types can be defined as $\mathrm{Cl}_{finite}[\alpha] :\equiv (\lambda x : itself\ \alpha.\ [\nexists f : nat \Rightarrow \alpha.\ injective[nat, \alpha]\ f\ ])$. The proposition $\mathrm{Cl}_c[\tau]\ (TYPE[\tau])$ serves as the logical interpretation of the type-in-class statement $\tau : c$ as defined in §2.1, and we abbreviate it by $(\!|\tau : c|\!)$.

When subclass relationships $c_1 \prec c_2$ and arities $\kappa :: (\overline{s_m})c$ are backed up by proofs of $\forall\alpha : \top.\ (\!|\alpha : c_1|\!) \Longrightarrow (\!|\alpha : c_2|\!)$ and $\forall\overline{\alpha_m} : \top.\ \overline{(\!|\alpha_m : s_m|\!)} \Longrightarrow (\!|\kappa\ \overline{\alpha_m} : c|\!)$, respectively, then $(\!|\tau : c|\!)$ is provable in Pure when $\tau : c$ holds. More precisely,

$$\tau : c \quad \text{implies} \quad \{(\!|\alpha^{\top} : c'|\!) \mid \alpha^s \text{ occurs in } \tau,\ c' \in s\} \vdash (\!|\tau^{\top} : c|\!)\ ,$$

where $\tau^{\top}$ denotes the type τ with all sort annotations replaced by $\top$.

To reflect this connection between the type system and the inference system, Pure provides a special proof constructor $\mathbf{ofclass}\,\tau\,c$ and the rule

$$\frac{\tau : c}{\Gamma \vdash \mathbf{ofclass}\,\tau\,c : (\!|\tau : c|\!)}\ .$$

The **ofclass** constructor serves as a placeholder for an explicit proof of $(\!|\tau : c|\!)$, which can always be constructed in a straightforward manner, following the rules for $\tau : c$.

Elimination of classes. To eliminate classes from propositions, we remove all sort annotations from type variables and replace them by explicit assumptions. Thus, a proposition $\forall\overline{\alpha_m : s_m}.\ \phi$ becomes $\forall\overline{\alpha_m : \top}.\ \overline{(\!|\alpha_m : s_m|\!)} \Longrightarrow \phi$.

Example 4. The proposition

$$\forall\alpha : \{finite\}.\ \bigwedge f : \alpha \Rightarrow \alpha.\ [\, injective[\alpha, \alpha]\ f \longleftrightarrow surjective[\alpha, \alpha]\ f\,]$$

is converted to

$$\forall\alpha : \top.\ (\!|\alpha : finite|\!) \Longrightarrow \bigwedge f : \alpha \Rightarrow \alpha.\ [\, injective[\alpha, \alpha]\ f \longleftrightarrow surjective[\alpha, \alpha]\ f\,]\ .$$

Eliminating classes from proofs amounts to this explication of sort constraints on type variables and replacing the proof constructors $\mathbf{ofclass}\,\tau\,c$ with derivations for $(\!|\tau : c|\!)$, using the newly-introduced assumptions on type variables.

A subtlety arises when the proof of a proposition contains type variables that do not occur in the proposition itself. Since the presence of such a type variable β^s constitutes an implicit assumption that the sort s is inhabited (i.e., $(\!|\tau : s|\!)$ holds for some ground type τ), we must introduce an extra hypothesis $(\!|\alpha : s|\!)$ for some canonical α. These sort inhabitedness hypotheses are tracked by Isabelle's inference kernel, ensuring soundness even when proof term recording is disabled. Term variables do not need a corresponding treatment, since types are always inhabited.

5.2 Overloading

Overloaded constants can have multiple defining equations on different types. To simplify the presentation, we assume that types of constants have exactly one parameter α, which allows us to write $c[\tau]$ instead of $c[\overline{\tau_m}]$. The treatment below is easily extended to the general case.

An overloaded constant c is specified by giving its type τ and multiple defining equations $c[\tau_i] :\equiv t_i : \tau[\alpha := \tau_i]$ for different type instances τ_i, where all type variables in the closed t_i have to occur in $\tau[\alpha := \tau_i]$, or equivalently in τ_i.

We write $c[\tau] \rhd d[\sigma]$ iff there exists a type τ_i, a substitution θ, and a defining equation $c[\tau_i] :\equiv t_i$, such that $\tau = \theta(\tau_i)$ and t_i contains a constant occurrence $d[\sigma']$ where $\sigma = \theta(\sigma')$. This relation on constants with types is called the *dependency relation*.

A system of overloaded definitions is well-formed, if the defining equations for any constant do not overlap (i.e., different τ_i and τ_j are not unifiable after renaming variables apart) and the dependency relation $\rhd$ is terminating. The latter property ensures that unfolding definitions cannot lead to non-termination and is undecidable [13], but Isabelle approximates this by a simpler criterion [7].

Note that this notion of overloading is more than just the use of a single name for multiple logical constants: The definition of another constant $d[\alpha]$ may refer to an overloaded constant $c[\tau]$, with the effect that the meaning of d also depends on instantiations of α. Then d is called *indirectly overloaded*. We call a constant $c[\tau]$ *overloading-free* iff it is primitive or has exactly one defining equation, whose right-hand side mentions only overloading-free constants.

A constant occurrence $c[\tau]$ in a term is called *resolvable* iff $\tau = \theta(\tau_i)$ for some substitution θ and some τ_i from a defining equation $c[\tau_i] :\equiv t_i$. Since defining equations do not overlap, τ_i and θ are then uniquely defined and we say $c[\tau]$ is *resolvable via* θ on $c[\tau_i]$.

Eliminating Overloading. The idea behind the elimination of overloading resembles the dictionary construction used to eliminate type classes from Haskell programs. For overloaded constants c, an overloading-free *dictionary constant* c_i is defined for each of the equations $c[\tau_i] :\equiv t_i$, abstracting out unresolvable overloading in t_i.

Concrete occurrences $c[\tau]$ can then be replaced by so-called dictionary terms: If $c[\tau]$ is resolvable, the corresponding dictionary constant is used, possibly passing through other dictionaries. If $c[\tau]$ is not resolvable, a dictionary variable $D_{c[\tau]}$ is inserted.

Formally, for a constant c and type τ, we define the set

$$\mathsf{dicts}(c[\tau]) := \begin{cases} \bigcup_{c[\tau] \rhd d[\tau']} \mathsf{dicts}(d[\tau']) & \text{if } c[\tau] \text{ is resolvable.} \\ \{c[\tau]\} & \text{otherwise.} \end{cases}$$

This set is well-defined and finite, since the dependency relation $\rhd$ is terminating and finitely branching. Lifting this to arbitrary terms, we define $\mathsf{dicts}(t) := \bigcup_{i \in \{1,\ldots,k\}} \mathsf{dicts}(d_i[\sigma_i])$, where $\overline{d_k[\sigma_k]}$ are the occurrences of constants in t. We assume some canonical order on $\mathsf{dicts}(t)$.

To eliminate overloading from a term, the mapping $[\![\cdot]\!]_{\mathsf{ov}}$ replaces (indirectly) overloaded constant occurrences $c[\tau]$ with dictionaries. Overloading-free constants and the rest of the term structure is preserved. If $c[\tau]$ is not resolvable, a dictionary variable is inserted.

$$[\![\, c[\tau] \,]\!]_{\mathsf{ov}} := D_{c[\tau]}$$

If $c[\tau]$ is resolvable via θ on $c[\tau_i]$, the corresponding dictionary constant c_i is used, passing dictionaries through:

$$[\![\, c[\theta\,\tau_i] \,]\!]_{\mathsf{ov}} := c_i[\theta\,\overline{\alpha_m}]\ \overline{[\![\, d_k[\theta\,\sigma_k] \,]\!]_{\mathsf{ov}}}\ ,$$

where $\{\overline{d_k[\sigma_k]}\} = \mathsf{dicts}(c[\tau_i])$ and $\overline{\alpha_m}$ are the type variables in τ_i.

The definitions of the dictionary constants c_i arise from the defining equations $c[\tau_i] :\equiv t_i$:

$$c_i[\overline{\alpha_m}] :\equiv (\lambda\, \overline{D_{d_k[\sigma_k]}}.\ [\![\, t_i \,]\!]_{\mathsf{ov}})$$

where $\overline{\alpha_m}$ are the type variables in τ_i and $\{\overline{d_k[\sigma_k]}\} = \mathsf{dicts}(c[\tau_i])$.

At the outermost level of propositions, we abstract over the generated dictionary variables.

$$[\![\, \forall\overline{\alpha_m}.\ \phi \,]\!]_{\mathsf{ov}} \ :=\ \forall\overline{\alpha_m}.\ \bigwedge \overline{D_{d_k[\sigma_k]}}.\ [\![\, \phi \,]\!]_{\mathsf{ov}} \qquad \text{where } \{\overline{d_k[\sigma_k]}\} = \mathsf{dicts}(\phi)\ .$$

Proofs are structurally unchanged, but constant definitions of overloaded constants are replaced by theorems about the resulting dictionary term.

Example 5. Assume an infix type constructor $\times$ with pair syntax $(\cdot,\cdot)$ and projections *fst* and *snd*, and a type *nat* where *nat-plus* : $nat \Rightarrow nat \Rightarrow nat$ defines addition. An overloaded addition function $plus : \alpha \Rightarrow \alpha \Rightarrow \alpha$ could be defined by the following equations.

$$\begin{aligned} plus[nat] &:\equiv nat\text{-}plus \\ plus[\alpha \times \beta] &:\equiv \lambda\, x\, y : \alpha \times \beta.\ (plus[\alpha]\ (fst\ x)\ (fst\ y), plus[\beta]\ (snd\ x)\ (snd\ y)) \end{aligned}$$

The overloading elimination introduces constants

$$\begin{aligned} plus_1 &:\equiv nat\text{-}plus \\ plus_2[\alpha,\beta] &:\equiv (\lambda\, (D_{plus[\alpha]} : \alpha \Rightarrow \alpha \Rightarrow \alpha)\, (D_{plus[\beta]} : \beta \Rightarrow \beta \Rightarrow \beta)\, (x\, y : \alpha \times \beta). \\ &\qquad (D_{plus[\alpha]}\ (fst\ x)\ (fst\ y), D_{plus[\beta]}\ (snd\ x)\ (snd\ y))) \end{aligned}$$

from which dictionaries for *plus* on arbitrarily nested tuples can be built, e.g.,

$$[\![\, plus[(nat \times nat) \times nat] \,]\!]_{\mathsf{ov}} = plus_2[nat \times nat, nat]\ (plus_2[nat, nat]\ plus_1\ plus_1)\ plus_1\ .$$

Notice the dictionary variables in the following translation of a simple theorem.

$$\begin{aligned} &[\![\, \forall\alpha\,\beta : \top.\ [\, comm[\alpha]\ plus[\alpha]\,] \Longrightarrow [\, comm[\beta]\ plus[\beta]\,] \Longrightarrow [\, comm[\alpha \times \beta]\ plus[\alpha \times \beta]\,] \,]\!]_{\mathsf{ov}} \\ &= \forall\alpha\,\beta : \top.\ \bigwedge (D_{plus[\alpha]} : \alpha \Rightarrow \alpha \Rightarrow \alpha)\, (D_{plus[\beta]} : \beta \Rightarrow \beta \Rightarrow \beta). \\ &\qquad [\, comm[\alpha]\ D_{plus[\alpha]}\,] \Longrightarrow [\, comm[\beta]\ D_{plus[\beta]}\,] \\ &\qquad\quad \Longrightarrow [\, comm[\alpha \times \beta]\ (plus_2[\alpha,\beta]\ D_{plus[\alpha]}\ D_{plus[\beta]})\,] \end{aligned}$$

Overloaded constants appearing in representing sets of type definitions are replaced in the same way using $[\![\cdot]\!]_{\mathsf{ov}}$. However, a fine point should be noted here: Abstracting out unresolvable overloading would give rise to dependent types. This is no problem in the set-theoretic interpretation, but as the overloading elimination is currently implemented as a preprocessing step on the HOL side, it does not handle such overloading. This can be fixed by collapsing the different parts of the translation. However, to remove unresolvable overloading from type definitions while staying in HOL, one has to eliminate the type definition altogether, replacing it by its representing type together with a predicate.

It appears that this subtle issue was overlooked in all proof sketches of conservativity of overloading so far [13,19,7]. Practically, this form of overloading seems to be quite rare. It does not occur in the main HOL image, but a few instances exist in the HOLCF development [12].

6 Discussion and Limitations

We briefly discuss some limitations of our approach and current implementation.

6.1 Replacing Constants and Types

The translations of some concepts defined in HOL are not as one would like to use them in set theory. For example, the translation $[\![int]\!]$ of the HOL integers should be the set $\mathbb{Z}$, which is already defined in Isabelle/ZF but happens to be a different (though isomorphic) object.

This problem is common to all proof translation tools and there is no general solution yet, apart from configuration to match up the concepts with equivalent ones. For example, in the proof translation from HOL4 and HOL Light to Isabelle/HOL [14], concepts can be replaced with others that behave in the same way, possibly with minor modifications such as argument order. In principle, our translation supports such replacements, but currently this requires tedious manual configuration and equivalence proofs.

6.2 Interactions of Pure and HOL

Recall that our translation inserts explicit constants *Lambda* and ‘ for abstraction and application in object-logic statements but leaves the outer proposition structure consisting of $\bigwedge$ and $\Longrightarrow$ intact. The advantage of this approach is that the results adhere to Isabelle's standard rule format. But in a few corner cases, the translation becomes difficult. For example, consider the following substitution rule for Pure equality:

$$\forall \alpha\ \beta : \top.\ \bigwedge (f : \alpha \Rightarrow \beta)\ (x : \alpha)\ (y : \alpha).\ x \equiv y \Longrightarrow f\ x \equiv f\ y$$

This is clearly a rule of the framework, as it contains no connectives of any object-logic. The translation should thus keep the rule as it is. On the other hand,

when α and β are instantiated with HOL types, then it should turn the types into sets and translate the application $f\ x$ to $f\text{`}x$. This shows that separating the framework from the object-logic cannot be done in a modular way. We have solved this problem by producing several variants for such rules, for different type instantiations.

An alternative translation that we would like to explore in the future is to abandon the distinction between Pure and HOL connectives, mapping everything to sets. Of course, dependencies on types must still use the framework, as type constructors and polymorphic constants are not sets in our model.

6.3 Performance

In its current implementation, the translation is expensive both in terms of time and memory. We exercised it on the main HOL image and HOL's number theory. Translating Fermat's little theorem and all its dependencies from the basic axioms takes 80 minutes and 1.6 GB of main memory on stock hardware. Measurements indicate that this performance cost is mainly due to extra $\beta\eta$-normalization of terms and type checking proofs becoming explicit in ZF. Obviously, more work is needed here to improve the performance.

7 Conclusion

Our translation maps all Isabelle/HOL primitives to set theory. By translating the proofs along with the theories, we can guarantee soundness of the overall method. The fact that we uncovered a notable omission in all previous proofs of conservativity of overloading (see §5.2) shows that our approach of "implemented semantics" is also useful for better understanding the logical system. Moreover, having an implementation facilitates experiments and modifications and will hopefully stimulate the further development of Isabelle/ZF.

Our elimination of type classes and overloading can also be of help when translating Isabelle/HOL developments to other systems. Previously, these concepts could only be translated by using extra-logical abstraction mechanisms provided by the OCaml language [10].

Acknowledgements. We would like to thank Stefan Berghofer, Florian Haftmann, and Makarius Wenzel, who patiently answered questions about the interior of the Isabelle system. Sascha Böhme and Armin Heller read a draft of this paper on short notice and made helpful comments.

References

1. Bortin, M., Broch Johnsen, E., Lüth, C.: Structured formal development in Isabelle. Nordic Journal of Computing 13, 1–20 (2006)
2. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation 76(2-3), 95–120 (1988)

3. Furbach, U., Shankar, N. (eds.): IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
4. Gaifman, H.: Global and local choice functions. Israel Journal of Mathematics 22 (3-4), 257–265 (1975)
5. Gordon, M.J.C.: Set theory, higher order logic or both? In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 191–201. Springer, Heidelberg (1996)
6. Gordon, M.J.C.: Twenty years of theorem proving for HOLs: Past, present and future. In: Ait Mohamed, O., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 1–5. Springer, Heidelberg (2008)
7. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 160–174. Springer, Heidelberg (2007)
8. Homeier, P.V.: The HOL-omega logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 244–259. Springer, Heidelberg (2009)
9. Lamport, L., Paulson, L.C.: Should your specification language be typed? ACM Transactions on Programming Languages and Systems 21(3), 502–526 (1999)
10. McLaughlin, S.: An interpration of Isabelle/HOL in HOL Light. In: Furbach, Shankar: [3], pp. 192–204
11. Moschovakis, Y.N.: Notes on Set Theory. Springer, Heidelberg (1994)
12. Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF=HOL+LCF. Journal of Functional Programming 9(2), 191–223 (1999)
13. Obua, S.: Checking conservativity of overloaded definitions in higher-order logic. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 212–226. Springer, Heidelberg (2006)
14. Obua, S., Skalberg, S.: Importing HOL into Isabelle/HOL. In: Furbach, Shankar: [3], pp. 298–302
15. Paulson, L.C.: The foundation of a generic theorem prover. Journal of Automated Reasoning 5, 363–397 (1989)
16. Paulson, L.C.: Set theory for verification: I. From foundations to functions. Journal of Automated Reasoning 11, 353–389 (1993)
17. Pitts, A.: The HOL logic. In: Gordon, M., Melham, T. (eds.) Introduction to HOL: A theorem proving environment for Higher Order Logic, pp. 191–232. Cambridge University Press, Cambridge (1993)
18. Schmidt-Schauß, M. (ed.): Computational Aspects of an Order-Sorted Logic with Term Declarations. LNCS, vol. 395. Springer, Heidelberg (1989)
19. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275. Springer, Heidelberg (1997)
20. Wiedijk, F.: The QED manifesto revisited. In: Matuszewski, R., Zalewska, A. (eds.) From Insight To Proof – Festschrift in Honour of Andrzej Trybulec, pp. 121–133. University of Białystok (2007)

The Isabelle Collections Framework

Peter Lammich[1] and Andreas Lochbihler[2]

[1] Universität Münster
peter.lammich@uni-muenster.de
[2] Karlsruher Institut für Technologie
andreas.lochbihler@kit.edu

Abstract. The Isabelle Collections Framework (ICF) provides a unified framework for using verified collection data structures in Isabelle/HOL formalizations and generating efficient functional code in ML, Haskell, and OCaml. Thanks to its modularity, it is easily extensible and supports switching to different data structures any time. For good integration with applications, a data refinement approach separates the correctness proofs from implementation details. The generated code based on the ICF lies in better complexity classes than the one that uses Isabelle's default setup (logarithmic vs. linear time). In a case study with tree automata, we demonstrate that the ICF is easy to use and efficient: An ICF based, verified tree automata library outperforms the unverified Timbuk/Taml library by a factor of 14.

1 Introduction

Isabelle/HOL [15] is an interactive theorem prover for higher order logic. Its code generator [7] extracts (verified) executable code in various functional languages from formalizations. However, the generated code often suffers from being prohibitively slow. Finite sets and maps are represented by chains of pointwise function updates, whose memory usage and run time are unacceptable for larger collections in practice. For example, to obtain an operative implementation, de Dios and Peña manually edited their generated code such that it used a balanced-tree data structure from the Haskell library [5, Sec. 5]. Not only are such manual changes cumbersome and error-prone as they must be redone each time the code is generated, they in fact undermine the trust obtained via formal verification.

There are some Isabelle/HOL formalizations of efficient collection data structures such as red-black trees (RBT), AVL trees [16], and unbalanced binary-search trees [11], each providing its own proprietary interface. This forces the user to chose the data structures at the start of formalization, and severely hinders switching to another data structure later. Moreover, wherever efficient data structures replace the standard types for sets and maps, one runs the risk of cluttering proofs with details from the data structure implementation, which obfuscates the real point of the proof. Furthermore, ad-hoc implementations of efficient data structures are scattered across other projects, thus limiting code reuse. For example, Berghofer and Reiter implemented tries for binary strings (called BDDs there), within a solver for Presburger arithmetic [2].

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 339–354, 2010.

This paper presents the Isabelle Collections Framework (ICF) that addresses the above problems. The main contribution is a unified framework (Sec. 2) to define and use verified collection data structures in Isabelle/HOL and extract verified and efficient code. As it works completely inside the logic, it neither increases the trusted code base, nor does it require editing the extracted code. The ICF integrates both existing (red-black trees, associative lists) and new (hashing, tries, array lists) formalizations of collection data structures. It provides a unified abstract interface that is sufficient for defining and verifying algorithms – independently of any concrete data structure implementation. This permits to change the actual data structure at any point without affecting the correctness proofs. To easily integrate existing data structures, the ICF contains a library of generic algorithms that implement most operations from a few basic operations. The ICF uses a data refinement approach that transfers the correctness statements from the abstract specification level to the concrete data structure implementation; Sec. 3 contains a small, but non-trivial example. With this approach, the ICF integrates well in existing formalizations.

Another contribution is our evaluation of the ICF (Sec. 4): (i) To benchmark its performance, we compared the ICF to the standard code generator setup and to library data structures of Haskell, OCaml, and Java. (ii) To demonstrate its usability in a case study, we implemented a formally verified tree-automata library [13] based on the ICF, using the data refinement approach. The ICF based tree-automata library outperforms the OCaml-based Timbuk/Taml library [6] by a factor of 14 and is competitive with the Java library LETHAL [14].

The ICF is published electronically in the *Archive of Formal Proofs* [12]. As the AFP is only updated with new Isabelle releases, a more recent version may be available at `http://cs.uni-muenster.de/sev/projects/icf/`.

1.1 Related Work

Most interactive theorem provers provide some efficient data structures in their libraries. The Coq standard library [4] features a modular specification for maps and sets that mirrors OCaml's library except for iterators. There are implementations based on strictly ordered (association) lists and on AVL trees for both sets and maps. There is also a trie implementation for maps with binary strings as keys. Coq's type system and code extraction facility allow the inclusion of data structure invariants (orderedness for lists and the search tree property for AVL trees) in the type definition without losing the capability to generate code. At present, Isabelle does not support this, i.e., the data structure invariants must be carried through all theorems explicitly. For ACL2 [10], there is a set implementation based on ordered lists, too.

For Haskell, Peyton Jones [17] proposes an elegant collections framework that uses type constructor classes and multi-parameter constructor classes. Unfortunately, Isabelle's type system supports neither of them.

The C++ Standard Template Library (STL) [18] provides the abstract concepts for the ICF: concepts (= ADTs), container classes (= implementations), algorithms (= generic algorithm), and iterators. In the STL, iterators are first-class

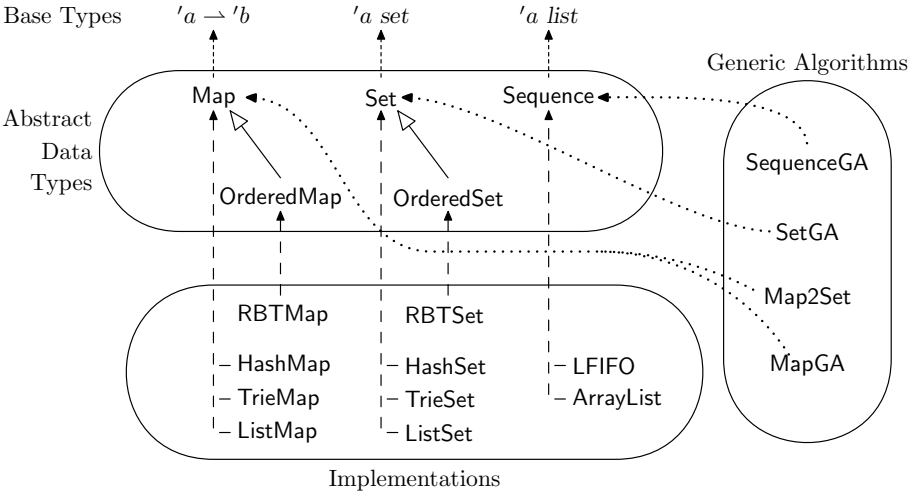


Fig. 1. The structure of the Isabelle Collections Framework

objects that represent the state of an iteration. In ICF, iterators are realized as combinators. While the first approach is more general (e.g., it allows one to iterate simultaneously over multiple data structures), ours is more convenient for use with a functional language. In C++, the compiler instantiates the templates automatically. This is not possible in Isabelle/HOL. Instead, the ICF contains (automatically generated) explicit instantiations of the generic algorithms. The user has to select the appropriate instantiation, which is easy due to a uniform naming scheme.

As for Java, the Java Collections Framework [9] provides an object-oriented approach. Interfaces describe ADTs. Concrete data structures implement them in classes. Generic algorithms are provided by means of static methods (in the `java.util.Collections` class) and abstract collection classes, which provide default implementations for most operations based on just a few basic ones (e.g. `java.util.AbstractSet`). Dynamic dispatch takes the role of instantiating the generic algorithms.

2 Overview of the Framework

Figure 1 outlines the structure of the Isabelle Collections Framework (ICF). Its main components are *abstract data types* (Sec. 2.1), *generic algorithms* (Sec. 2.4), and *implementations* (Sec. 2.6). An abstract data type (ADT) specifies a set of operations and their behavior (e.g. a set with empty, member, insert, delete and iteration operations) w.r.t. a base type in Isabelle (Sec. 2.2). ADTs can extend other ADTs, denoted by solid lines. An implementation provides an actual data structure with operations (dashed lines), and proves that they match the specification of the ADT (e.g. HashSet implements the ADT set). A generic algorithm

implements some operation via other ADT operations (dotted lines) – independently of the concrete implementation. For this, the *iterators* (Sec. 2.3) that each ADT provides are extremely useful.

2.1 Abstract Data Types

The ICF currently supports three different abstract data types:

Maps. A map is a partial function from keys to values with finite domain. The operations include the empty map constructor, emptiness check, lookup, update, deletion, composition, iteration, conversion to associative lists, and quantification and choice over the domain. Ordered maps extend maps in that they require a linear order on the keys and provide operations to iterate over the keys in ascending and descending order. Currently, there are map implementations using association lists, red-black trees (RBT), hashing, and tries. The RBT implementation is also an ordered map.

Sets. A finite set with empty set constructor, insertion, intersection, union, difference, emptiness, membership and subset checks, cardinality, iteration, image, and choice operations. Like for maps, the ICF specifies ordered sets that require a linear order on the elements and provide ordered iteration. Except for ListSet, the set implementations are derived from the corresponding map implementations via the generic algorithm Map2Set.

Sequences. A finite sequence. Unlike sets and maps, the insertion order determines the iteration order. Implementations are a queue LFIFO with amortized constant-time *enqueue* and *dequeue* as well as *push* and *pop* operations, and a resizable array implementation that provides access by index positions.

2.2 Data Refinement

The operations of an ADT are specified by its intended behavior w.r.t. an abstraction mapping α that abstracts from the concrete implementation to the so called *base type* of the ADT. The base type of sets is Isabelle/HOL's type $'a\ set$, maps have the base type $'k \to 'v\ option$[1], and sequences have the base type $'a\ list$. For example, the *empty*, *memb*, and *ins* operations of the ADT set are specified as follows:[2]

$$\begin{aligned}&\mathsf{empty\text{-}correct}: \alpha\ empty = \{\}\\ &\mathsf{memb\text{-}correct}: memb\ x\ s \Leftrightarrow x \in (\alpha\ s)\\ &\mathsf{ins\text{-}correct}: \quad \alpha\ (ins\ x\ s) = \{x\} \cup (\alpha\ s)\end{aligned}$$

A proposition that involves ADT operations is usually proved in two steps:

1. Transform it into a proposition that only involves operations on the base type. This is straightforward using the specifications of the ADT's operations and usually done automatically by Isabelle's simplifier.

[1] The data type $'a\ option = None \mid Some\ 'a$ corresponds to `Maybe a` in Haskell.

[2] To simplify the presentation, we omitted the data structure invariant *invar*, which guards all specifications for abstract data types (cf. Sec. 2.5).

2. Prove the transformed proposition. Since it only involves base types, this proof enjoys the full support of Isabelle's automated proof methods and is completely independent of the ICF.

For example, the proposition *memb y* (*ins x* (*ins y* (*ins z empty*))) is first transformed to $y \in \{x, y, z\}$ and then proved automatically by the method auto.

Hence, when using the ICF to implement some algorithm, the algorithm is first formalized and proved correct on the base type – independently of the ICF. In a second (straightforward) step, definitions are transformed to use the ADTs of the ICF, and correct data refinement is shown. This approach also simplifies porting existing formalizations to the ICF, which requires only the second step, while the existing correctness proofs remain untouched. Section 3 presents an example of this approach in detail.

2.3 Iterators

Iterators are one of the ICF's key concepts. An iterator over a finite set or map is a generalized *fold* combinator: It applies a state-transforming function to all elements of a set (entries of a map, resp.), starting with an initial state and returning the final state. Additionally, the iteration is interrupted if a continuation condition on the state no longer holds. The iteration order is unspecified.

The Isabelle/HOL standard library provides an uninterruptible *fold* combinator for finite sets that requires the state-transformer to be left-commutative[3] to ensure that the iteration result does not depend on the iteration order. However, generic algorithms in the ICF typically use state-transformers that are not left-commutative. Consider, e.g., a generic algorithm for coercion between set implementations. It iterates over the source set and inserts each element into the target set which is initially empty. Although inserting is left-commutative on the base type, the shape of the target set's data structure usually depends on the insertion order, i.e. inserting is *not* left-commutative for data structures. Hence, ICF iterators do not require left-commutativity.

An iterator *iterate* on a set data structure of type $'s$ has the type $('\sigma \to \mathit{bool}) \to ('a \to '\sigma \to '\sigma) \to '\!s \to '\sigma \to '\sigma$. It takes a continuation condition c, a state transformer f, the set s, and the initial state σ. For reasoning, the following rule is used, which requires an iteration invariant I:

$$[\text{iterate-rule}]\quad \frac{\begin{array}{c} I\ (\alpha\ s)\ \sigma_0 \\ \forall x\ i\ \sigma.\ c\ \sigma \wedge x \in i \wedge i \subseteq \alpha\ s \wedge I\ i\ \sigma \Longrightarrow I\ (i - \{x\})\ (f\ x\ \sigma) \\ \forall \sigma.\ I\ \{\}\ \sigma \Longrightarrow P\ \sigma \\ \forall \sigma\ i.\ i \subseteq \alpha\ s \wedge i \neq \{\} \wedge \neg c\ \sigma \wedge I\ i\ \sigma \Longrightarrow P\ \sigma \end{array}}{P\ (\mathit{iterate}\ c\ f\ s\ \sigma_0)}$$

The iteration invariant $I :: '\!x\ \mathit{set} \to '\sigma \to \mathit{bool}$ takes two parameters: (i) the iteration state i denotes the set of elements that still needs to be iterated over, and (ii) the computed state σ of type $'\sigma$. To establish a property P of the resulting state, it must be shown that:

[3] A function $f :: 'a \to '\sigma \to '\sigma$ is called *left-commutative* iff $\forall x\ y.\ f\ x \circ f\ y = f\ y \circ f\ x$.

1. the iteration invariant I holds for the initial state σ_0 with the whole set $\alpha\ s$ unprocessed,
2. the state transformer f preserves I for any removal of any element from any subset of $\alpha\ s$, and
3. $I\ i\ \sigma$ implies $P\ \sigma$, when the iteration stops either normally ($i = \{\}$) or prematurely ($\neg c\ \sigma$, $i \neq \{\}$).

Example 1. The following algorithm *copy* copies a set from a source implementation (indexed 1) to a target implementation (indexed 2).

$$copy\ s_1 = iterate_1\ (\lambda\sigma.\ True)\ ins_2\ s_1\ empty_2$$

The continuation condition ($\lambda\sigma.\ True$) ensures that iteration does not stop prematurely. The iteration state is the data structure of the target implementation. The initial state is the empty set $empty_2$, and the state transformer function is the insert function ins_2 of the target implementation. To prove *copy* correct – i.e. if $invar_1\ s_1$, then $invar_2\ (copy\ s_1)$ and $\alpha_2\ (copy\ s_1) = \alpha_1\ s_1$ – we use the iteration invariant $I\ i\ s_2 = (\alpha_2\ s_2 = (\alpha_1\ s_1) - i \wedge invar_2\ s_2)$.

Example 2. Bounded existential quantification ($\exists x \in s.\ P\ x$) can be implemented via iteration: $bex\ s\ P = iterate\ (\lambda\sigma.\ \neg\sigma)\ (\lambda x\ \sigma.\ P\ x)\ s\ False$. The state of this iteration is a Boolean that becomes true when the first element satisfying P is found. The iteration stops prematurely when the state becomes true.

For proving *bex* correct – i.e. if $invar\ s$, then $bex\ s\ P = (\exists x \in \alpha\ s.\ P\ x)$ – we use the iteration invariant $I\ i\ \sigma = (\sigma = (\exists x \in (\alpha\ s) - i.\ P\ x))$.

2.4 Generic Algorithms

A generic algorithm implements and proves correct a *target operation* by means of a set of *source operations*, independently from the actual data structure. To obtain an implementation of the target operation together with its correctness statement, the generic algorithm and its correctness statement are instantiated with actual implementations of the source operations.

Generic algorithms reduce redundancy because an algorithm needs to be proved correct only once and is then instantiated for various implementations of the involved ADTs. For example, the *map-to-nat* function computes a bijective map from a finite set into an initial segment of the natural numbers. It is defined and proved correct independently of the actual map and set implementations.

Generic algorithms are also used to reduce the effort of creating a new implementation. The ADTs provide generic algorithms to derive most operations from a small set of basic operations, using iterators as a key concept. For example, all specified map and set operations can be implemented by iterators and four basic operations: the empty map or set constructor, lookup or membership test, insertion, and deletion. In a later development stage, these generic implementations may be replaced by versions optimized for the actual data structure. However, as the generic algorithms are reasonably efficient, this is often not necessary. As a special case, the ICF contains generic algorithms to derive a set implementation from a map implementation by using a map with value type *unit*, whose only element is ().

2.5 Realization within Isabelle/HOL

In this section, technical details and challenges of the ICF's realization within Isabelle/HOL are discussed.

Abstract Data Types. The ICF uses Isabelle/HOL's locale mechanism[4] [1] to specify an ADT. For each ADT, a base locale fixes the data structure invariant and the abstraction function. Each operation is specified by its own locale, which extends the base locale, fixes the operation and specifies its behavior. For example, the following locales specify the ADT set and its delete operation:

locale *set* = **fixes** $\alpha :: 's \to 'a\ set$ **and** $invar :: 's \to bool$

locale *set-delete* = *set* +
  fixes $delete :: 'a \to 's \to 's$
  assumes delete-correct :
    $invar\ s \Longrightarrow \alpha\ (delete\ x\ s) = (\alpha\ s) - \{x\}$
    $invar\ s \Longrightarrow invar\ (delete\ x\ s)$

Note that the data structure invariant *invar* guards all specification equations to allow for ADT implementations that require invariants.

Implementations. An implementation interprets the locales with the operations it provides, thereby showing that they satisfy the ADT's specification. The *HashSet* implementation, e.g., defines the functions *hs-α*, *hs-invar*, and *hs-delete* and proves the lemma hs-delete-impl: *set-delete hs-α hs-invar hs-delete* where *set-delete* denotes the assumption predicate of the locale *set-delete*. From this lemma, interpretation produces the lemma hs.delete-correct, which is delete-correct with the parameters instantiated by *hs-α*, *hs-invar*, and *hs-delete*. Note that the dot (instead of a dash) in *hs.delete-correct* has technical reasons.

Naming Conventions. The ICF uses several naming conventions that simplify its usage: The locale specifying an operation *op* for an ADT *adt* is named *adt-op* (e.g., *set-delete*). The correctness assumption is called *op-correct* (e.g., *delete-correct*). Each implementation of an ADT has a short (usually two letters) prefix (e.g., *hs* for HashSet). An implementation with prefix *pp* provides a lemma *pp-op-impl* and interprets the operation's locale with the prefix *pp*, yielding the lemma *pp.op-correct*.

Data Refinement. The proof of the data refinement step is, in many cases, performed automatically by the simplifier. In some complex cases, involving, e.g., recursive definitions or nested ADTs, a small amount of user interaction is necessary. Section 3 contains an example for such a complex case.

[4] Locales provide named local contexts with fixed parameters (**fixes**) and assumptions (**assumes**). They support inheritance (+) and interpretation (i.e., parameter instantiation), which requires to discharge the assumptions. A predicate with the locale's name collects all assumptions of the locale.

Generic Algorithms. A generic algorithm is defined as a function that takes the source operations as arguments. The correctness lemma shows that the target operation meets its specification if the source operations meet theirs.

Example 3. Reconsider the bounded existential quantification from Ex. 2, where an *iterate* operation was used. In a generic algorithm, this operation becomes an additional parameter:

$$bex\ iterate\ s\ P = iterate\ (\lambda\sigma.\ \neg\sigma)\ (\lambda x\ \sigma.\ P\ x)\ s\ False$$

Assume that the locale *set-bex* specifies bounded existential quantification. Then, we prove the correctness lemma

$$\mathsf{bex\text{-}correct}\text{: } set\text{-}iterate\ \alpha\ invar\ iterate \Longrightarrow set\text{-}bex\ \alpha\ invar\ (bex\ iterate)$$

An instantiation then sets the parameters α, *invar*, and *iterate* to its operations. In Isabelle/HOL, this is easily done with the *OF* and *folded* attributes, as illustrated in the following example, that instantiates the algorithm for HashSets:

definition *hs-bex* = *bex hs-iterate*
lemmas *hs-bex-impl* = *bex-correct*[OF *hs-iterate-impl*, folded *hs-bex-def*]

where hs-iterate-impl: *set-iterate hs-α hs-invar hs-iterate*. Hence, we get the lemma hs-bex-impl: *set-bex hs-α hs-invar hs-bex*.

Unfortunately, Isabelle cannot generate instantiations of a generic algorithm automatically like Coq with its implicit arguments or C++ with its template mechanism. Instead, the ICF contains (automatically generated) explicit instantiations for each combination of generic algorithm and implementation, using a uniform naming scheme. It remains up to the user to select the appropriate instantiation, which is easy due to the uniform naming scheme. For example, to compute the union of a list-based set with a hash set, yielding a hash set, the user has to pick the function *lhh-union*, where the prefix *lhh* selects the right instantiation of the union-algorithm.

When implementing an algorithm using the ICF, the user must choose between writing a generic algorithm or fixing the data structures in advance. A generic algorithm needs to be parameterized over all used operations. If an algorithm uses only a few operations, the parameterization may be done explicitly, as in Ex. 3. If many different operations are involved, parameterization can be hidden syntactically in locale context, in order not to mess up the definitions with long parameter lists. However, due to restrictions in Isabelle/HOL's polymorphism, every collection with different element type requires its own operation parameters. Similarly, one has to specify one monomorphic instantiation for each iterator with different state. Alternatively, a record can collect all required ADT operations, but the monomorphism issue remains.

When a generic algorithm is not required, one can fix the used data structures beforehand, either by making alias definitions for the concrete operations and lemmas at the beginning of the theory, or by directly using the concrete

operations and lemmas throughout the theory. This avoids the above-mentioned problems with polymorphism. Thanks to the consistent naming conventions used in the ICF, switching to another implementation is as easy as replacing the prefixes of constant and lemma names (e.g., replacing *rs-* by *ts-* to switch from RBTs to Tries).

An alternative approach (similar to Peyton Jones' `XOps` route that automatically selects the data type implementation [17, Sec. 3]) is to hide ADT implementations completely from the ADT inside the logic. To that end, we introduce a new type for each ADT which is isomorphic to its base type. In the generated code, the new type becomes a data type with one constructor for each implementation. The abstract operations then pattern match on the ADT and dispatch to the correct implementation – emulating dynamic dispatch of the Java Collection Framework. Concrete implementations are selected by choice of the constructor, and, thanks to dynamic dispatch, manual instantiation or selection of generic algorithms is no longer necessary. However, this approach currently only works for ADT implementations that do not require invariants. Thus, it is only implemented for tries and array-based hashing. Yet, the Isabelle developers are working on the code generator such that it can handle such invariants.[5]

2.6 Implementations for ADTs

The ICF implementations for the ADTs use four basic data structures: lists, arrays, red-black trees, and tries. Inside Isabelle/HOL, arrays are isomorphic to lists, but for Haskell code, we use the `Data.Array.Diff.DiffArray` implementation from the Haskell library, which supports in-place updates while providing the immutable (functional) interface. To our knowledge, ML's and OCaml's standard libraries do not feature a similar implementation, so we fall back on a list-based implementation. Red-black trees are taken from the Isabelle/HOL standard library and extended with the iterator concept. A trie (prefix tree) is a search tree for strings where the key string identifies the path from the root to the node that stores the value. In contrast to RBTs, tries do not need data structure invariants. Our implementation improves upon and substantially extends the one in [15, Ch. 3.4.4]. The implementations for the ADTs use these data structures to implement maps, sets and sequences (cf. Fig. 1).

Maps. There are four implementations for finite maps: association lists (ListMap), red-black trees (RBTMap), hashing (HashMap, based on either RBTs or arrays), and tries (TrieMap).

Association lists have the data structure invariant that every key is unique in the list. While not being necessary, this allows for a simpler and more efficient implementation of iterators. Association lists work for all key types with executable equality test.

Red-black trees require that the key type is linearly ordered; the invariant ensures that it is a correct RBT, i.e., it has no two consecutive red nodes on a path, balanced height, the root is black, and the entries are ordered by their key.

[5] Personal communication with F. Haftmann.

If a key type does not have a canonical linear order, one can still use red-black trees by prefixing a hash operation *hashcode* that maps keys to integers. Then, the RBT maps an integer (the key's hashcode) to a bucket, which stores the key-value pairs for all keys with that hashcode in an association list. We use Isabelle's type classes to overload the *hashcode* function for different types and provide instantiations for all standard Isabelle type constructors except for functions (because they cannot be tested for equality). The invariant for hashing backed by RBTs is (i) the invariant for the RBT itself and (ii) that the keys in any bucket (i.e. association list) are distinct and have the bucket's hashcode.

The ICF also offers hashing backed by an array, which is currently only sensible with Haskell code (cf. above). This provides access in constant time, but requires to grow the array and rehash all data in the map when the load increases beyond a certain threshold. Our implementation triggers a rehash when the number of keys reaches 75% of the array size, a standard load factor threshold for open hashing.

By definition, keys for tries must be strings. For all other types, we use an encoding function *encode* into strings of integers, which must be injective. For natural numbers, e.g., we compute the 16-adic representation starting with the lowest digit, i.e. $1000 = 3 \cdot 16^2 + 14 \cdot 16 + 8$ is encoded as $[8, 14, 3]$. The type class for this encoding pairs every *encode* function with a left-inverse partial function *decode* that decodes the strings. Since *encode* is one-to-one, only countable types may be used as keys in a trie. Like for hashing, the ICF provides instantiations for all countable types predefined in Isabelle/HOL.

Sets. A map whose value type is the singleton type *unit* is isomorphic to a set, where mapping a key k to () means that the set contains k. The ICF provides generic algorithms (Map2Set) such that an implementation for the ADT map easily yields one for the ADT set. The RBT, hashing, and trie implementation for maps use this setup to define the set implementations.

Sequences. Sequences are typically used in two different styles: array-like and stack- or queue-like. In the array-like style, elements are accessed by index, and new elements are appended at the end. If implemented with a linked-list data type, these operations take linear time. The ICF therefore provides an array-based implementation ArrayList, which provides index and appending operations in amortized constant time (for Haskell), enlarging the array as necessary. However, prepending an element must shift all elements, which takes linear time.

In case a stack or queue is needed, the ICF contains an amortized constant-time queue implementation LFIFO that also provides constant-time stack operations. However, access by index is implemented by iteration, which takes linear time on average.

3 An Example Application

In this section, we demonstrate how to use the ICF in an example application inspired by the Sieve of Eratosthenes. Whereas the traditional Sieve produces a

$$
\begin{aligned}
\mathit{sieve}\ n &\equiv \mathit{sieve}_1\ n\ 2\ (\lambda\_.\ \{\}) \\
\mathit{sieve}_1\ n\ i\ M &\equiv \mathsf{if}\ n < i\ \mathsf{then}\ M \\
&\qquad \mathsf{else}\ \mathit{sieve}_1\ n\ (i+1)\ (\mathsf{if}\ M\ i = \{\}\ \mathsf{then}\ \mathit{addp}\ n\ i\ i\ M\ \mathsf{else}\ M) \\
\mathit{addp}\ n\ p\ j\ M &\equiv \mathsf{if}\ j > n\ \mathsf{then}\ M\ \mathsf{else}\ \mathit{addp}\ n\ p\ (j+p)\ (M(j := \{p\} \cup M\ j))
\end{aligned}
$$

Fig. 2. The modified Sieve of Eratosthenes to compute sets of prime divisors

list of prime numbers less than n, we produce a map from numbers less than n to their set of prime divisors, ignoring their multiplicity. A tail-recursive implementation is shown in Fig. 2, where *sieve* n runs the function sieve_1 for the first n numbers and returns a function M of type $\mathit{nat} \to \mathit{nat\ set}$ such that for all i within 2 and n, $M\ i$ is the set of all prime divisors of i. $\mathit{sieve}_1\ n\ i\ M$ iterates from i up to n and, whenever it encounters a new prime number i ($M\ i = \{\}$), it adds i to the set $M\ j$ of all multiples j of i up to n via the function *addp*.

However, this implementation is not executable, because it contains the test $M\ i = \{\}$ (i.e. a function equality[6]). One solution is to change M's type to $\mathit{nat} \to \mathit{nat\ set\ option}$ and replace $\{\}$ with *None*, but this is very inefficient because the function M is built from pointwise updates, i.e. a function application $M\ i$ takes time linear in the number of updates. Since the number of prime divisors $\omega(n)$ of n is in $\mathcal{O}(\log(\log(n)))$ [8], there are $\mathcal{O}(n \cdot \log(\log(n)))$ updates and the above application executes $\mathcal{O}(n)$ times. Since sets and maps are coded as functions, insertion and update only add function closures and therefore require constant time. Hence, the overall run time is in $\mathcal{O}(n^2 \cdot \log(\log(n)))$.

We now reformulate the sieve as a generic algorithm by replacing the map M and the sets in the range of M by ICF ADTs (and implementations). Note that the Sieve could be implemented much more efficiently using an array monad and lists instead of sets. Yet, it still is a good non-trivial example to illustrate how to integrate the ICF in one's formalization, because the set ADT is nested in the map ADT. Following the data refinement approach from Sec. 2.2, the Sieve integrates with the ICF in three steps:

1. For code generation, we define new functions that operate on ADTs (Fig. 3).
2. We show that the new functions preserve the data structure invariants.
3. We show transfer equations between original and new functions.

The equations proved in step 3 are then used to transfer correctness theorems from the original functions to the new functions.

The sieve is defined as a generic algorithm, i.e. the definitions are (implicitly) parameterized over the used operations. The implicit parameterization is achieved by combining the locales for maps and sets, thereby prefixing the map and set operations with *m-* and *s-* resp., to avoid name clashes. Fig. 3 shows the new implementation. Note that the algorithm is structurally the same, but the operations on sets and maps have been replaced by the ADT operations, for

[6] In Isabelle/HOL, a set is represented by its characteristic function.

$$
\begin{aligned}
&\mathit{sieve}'\ n \equiv \mathit{sieve}'_1\ n\ 2\ \mathit{m\text{-}empty}\\
&\mathit{sieve}'_1\ n\ i\ M \equiv \mathsf{if}\ n < i\ \mathsf{then}\ M\\
&\qquad \mathsf{else}\ \mathit{sieve}'_1\ n\ (i+1)\ (\mathsf{if}\ \mathit{m\text{-}lookup}\ i\ M = \mathit{None}\ \mathsf{then}\ \mathit{addp}'\ n\ i\ i\ M\ \mathsf{else}\ M)\\
&\mathit{addp}'\ n\ p\ j\ M \equiv \mathsf{if}\ n < j\ \mathsf{then}\ M\\
&\qquad \mathsf{else}\ \mathit{addp}'\ n\ p\ (j+p)\ (\mathit{m\text{-}update}\ j\ (\mathit{s\text{-}ins}\ p\ (\mathit{opt\text{-}dest}\ (\mathit{m\text{-}lookup}\ j\ M)))\ M)\\
&\mathit{opt\text{-}dest}\ \mathit{None} \equiv \mathit{s\text{-}empty} \qquad \mathit{opt\text{-}dest}\ (\mathit{Some}\ A) \equiv A
\end{aligned}
$$

Fig. 3. Implementation of the modified Sieve with the ICF

which no syntactic sugar is currently available. The new function *opt-dest* stems from replacing the function M by a map where *None* represents the empty set.

Since the ADTs are nested, their abstraction functions and invariant predicates are combined into new ones:

$$
\begin{aligned}
\alpha\ M &\equiv \mathit{s\text{-}\alpha}\ (\mathit{opt\text{-}dest}\ (\mathit{m\text{-}\alpha}\ M))\\
\mathit{invar}\ M &\equiv \mathit{m\text{-}invar}\ M \wedge (\forall n\ S.\ \mathit{m\text{-}\alpha}\ n = \mathit{Some}\ S \Longrightarrow \mathit{s\text{-}invar}\ S \wedge \mathit{s\text{-}\alpha}\ S \neq \{\})
\end{aligned}
$$

Note that we exclude empty sets being stored in M, because *None* already represents them. Next, we show that *addp'* and *sieve'* preserve the data structure invariant *invar*. This is straightforward because they only use the abstract operations that preserve the invariants by assumption. Finally, proving the following transfer equations is also straightforward under the assumption $\mathit{invar}\ M$:

$$\alpha\ (\mathit{addp}'\ n\ p\ M) = \mathit{addp}\ n\ p\ (\alpha\ M) \tag{1}$$

$$\alpha\ (\mathit{sieve}'_1\ n\ i\ M) = \mathit{sieve}_1\ n\ i\ (\alpha\ M) \tag{2}$$

$$\alpha\ (\mathit{sieve}'\ n) = \mathit{sieve}\ n \tag{3}$$

Since $\mathit{invar}\ (\mathit{sieve}'\ n)$ holds, $\mathit{m\text{-}lookup}\ j\ (\mathit{sieve}'\ n)$ returns the set of j's prime divisors for all $2 \leq j \leq n$.

To obtain an executable implementation with concrete data structures, e.g. RBTs, we simply interpret our locale. The ICF is set up such that all proof obligations are discharged automatically. For the run time complexity of the RBT implementation, the map operations dominate the set operations because the set size is limited by $\mathcal{O}(\log(\log(n)))$. Since M is updated $\mathcal{O}(n \cdot \log(\log(n)))$ times, the overall run time is in $\mathcal{O}(n \cdot \log(n) \cdot \log(\log(n)))$.

4 Evaluation

This section reports on some performance measurements. In Sec. 4.1, we compare the generated code using the ICF with the code generated from the Isabelle/HOL default set representation, and with the tree data structures from the standard libraries of Haskell and OCaml.[7] Then, we briefly describe a tree automata

[7] Unfortunately, the SML standard library contains no tree structure.

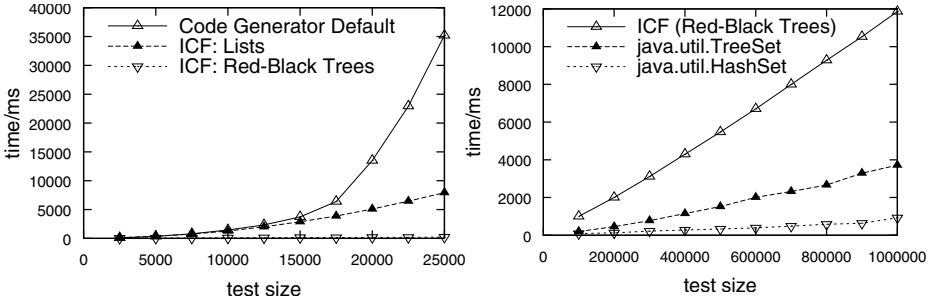


Fig. 4. Comparison of ICF data structures with code generator defaults and with Java

library that is based on the ICF and compare its performance to a well-known OCaml library and a Java library (Sec. 4.2).

All tests where done on a 2.66 GHz x86/64 dual-core machine with 4 GB of memory. We used Poly/ML 5.2, OCaml 3.09.3, GHC 6.10.4, and OpenJDK 1.6.0-b09. The run time values are averages over three test runs.

4.1 Basic Operations

For comparing the performance of basic set operations, we ran a simple program that starts with an empty set, then inserts n times a random number in the range $[0, 2n)$, then removes n times a random number in the range $[0, 2n)$, then tests n times a random number in the range $[0, 2n)$ for membership in the set, and finally iterates over each element in the set. As iteration is not executable in Isabelle/HOL's default code generator setup, we omitted the last phase when comparing with the default setup. This program exercises exactly the basic operations (*empty*, *member*, *insert*, *delete*, *iterate*) from which the other set operations may be (and actually are) derived by generic algorithms.

The left part of Figure 4 shows the runtimes of the code generated by Isabelle from a set-based formalization using the standard code generator setup and of the code that uses ICF data structures. The x axis shows the test size n and the y axis the required time in milliseconds. This test was done on the Poly/ML platform, which was faster than OCaml and GHC for this kind of tests. Clearly, the run time of the code generated from the default setup grows significantly faster (theoretically $\mathcal{O}(n^2)$) than the code using the ICF red-black trees (theoretically $O(n \log n)$). However, even the list-based ICF set implementation (also $O(n^2)$) is significantly faster than the default setup, because the latter's chain of pointwise function updates grows also with every delete operation.

The right part of Fig. 4 compares ICF's red-black trees to Java's `TreeSet` and `HashSet` classes. Java's `HashSet` class is backed by an array, and thus more efficient than the tree implementations, whose overhead per operation is much larger. Moreover, Java uses destructive updates whereas the ICF is purely functional. The ICF test was, again, run on the Poly/ML platform. Java's `TreeSet` is, on average, 3.7 times faster than the RBTs from the ICF – the ratio decreases

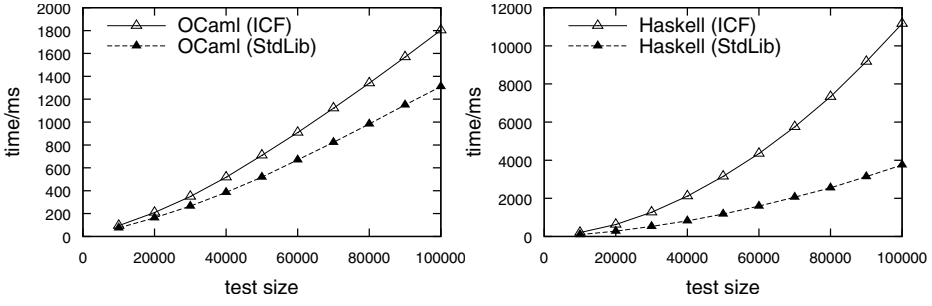


Fig. 5. Comparison of the ICF with the OCaml and Haskell standard library

from 5.0 for $n = 10^5$ to 3.2 for $n = 10^6$. Java's `HashSet` is even 15.2 times faster on average. These results show how much room there is for speed-up, when one is not bound to platform constraints.

In order to capture the potential for improvement on a functional language platform (to which the Isabelle/HOL code generator is restricted), we compare the red-black trees from the ICF with tree data structures from the Haskell and OCaml standard libraries. The results are shown in Fig. 5. For OCaml, the standard library is, on average, 34% faster than the ICF – the ratio increases from 25% for $n = 10^4$ to 38% for $n = 10^5$. For Haskell, the difference is even more significant. Here, the standard library is, on average, about 2.6 times faster – the value increases from 2.0 for $n = 10^4$ to 3.0 for $n = 10^5$. The significant super-linear increase for Haskell also results from lazily evaluated tail-recursive functions.[8]

Currently, the ICF tree data structure uses RBTs from the Isabelle standard library. Our results show that there is still room for improvement on the data structure's efficiency. On the other hand, the ICF data structures are formally verified, whereas those of the Haskell and OCaml standard libraries are not. Moreover, it would also be possible to configure the code generator to use the data structures from the standard library instead of the verified ones.

4.2 Case Study: An ICF-Based Tree Automata Library

The first author has implemented a formally verified tree automata library [13]. It uses the ICF to derive efficient code and the data refinement approach to verify algorithms on an implementation independent (and thus simpler) level. We compared the generated code to Timbuk/Taml [6], a tree automata library for OCaml, and to LETHAL [14], a tree automata library for Java that has been developed as a students' project in our group.

The test consisted of intersecting seven pairs of randomly generated tree automata (with a few hundred rules and up to one hundred states each), and

[8] Up to a certain extent, strict evaluation in Haskell can be forced by the *seq*-operator. However, we did not include such platform specific optimizations into the ICF.

Table 1. Tree Automata Library using the ICF compared to other libraries

Language	ICF Haskell	ICF SML	ICF OCaml	ICF OCaml(i)	Taml OCaml(i)	LETHAL Java
complete	1.5s	6.1s	12.5s	121s	1923s	0.456s
reduced	73ms	407ms	522ms	4983ms	71636ms	120ms

then checking the results for emptiness. Table 1 shows the run time for various platforms. All ICF versions used RBT-based hashing. Most notably the ICF library running on Haskell is three orders of magnitude faster than Timbuk/Taml. However, this mainly results from comparing compiled Haskell with interpreted OCaml (marked as OCaml(i) in the table header). Another issue is that the ICF-based library uses a different algorithm for checking emptiness that performs better for automata with non-empty languages. However, even when comparing the ICF-based library and Timbuk/Taml both on interpreted OCaml, with a reduced test set (second row) where the tested automata's languages are all empty, the ICF based library is still about 14 times faster. We conjecture that Timbuk/Taml's use of plain lists for sets and maps – which is common practice in functional programming – explains that difference.

For the complete test set, the Java-based LETHAL library is about three times faster than the ICF-based library running on Haskell. For the reduced test set, the latter is even a bit faster than the Java implementation. These encouraging results demonstrate that it is possible to use the ICF to develop efficient verified algorithms that are competitive with existing unverified ones.

5 Conclusion

The Isabelle Collections Framework is a unified, easy-to-use framework for using verified data structures in Isabelle/HOL formalizations. Abstract data types for common Isabelle types provide the option to generate efficient code for a wider class of operations than the default setup. Data refinement allows one to transfer correctness results from existing formalizations to efficient implementations by means of transfer equations. The ICF implementations vastly outperform the standard code generator setup. The ICF proved its usability and efficiency in a verified tree automata library: The generated code outperforms the well-known (unverified) Timbuk/Taml library by a factor of 14, and is even competitive with the Java-based (also unverified) LETHAL library.

However, a lot remains to be done. The evaluation shows that the data structures are not yet optimally efficient. Some data structures, like heaps and priority queues, are still missing. Concerning usability, Isabelle's code generator currently poses the biggest limitation. When it will support invariants for data types, the ICF will integrate much more smoothly into existing formalizations.

Another approach to make functional code more efficient are state monads that support, e.g., arrays with destructive updates. The Imperative HOL framework [3] adds support for monads to Isabelle/HOL. It remains future work to implement and verify monadic collection data structures.

Acknowledgement. We thank Nicholas Kidd and Alexander Wenner for proofreading and many helpful comments.

References

1. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 31–43. Springer, Heidelberg (2006)
2. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: TPHOLs '09, pp. 147–163. Springer, Heidelberg (2009)
3. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008)
4. The Coq standard library, http://coq.inria.fr/stdlib/index.html
5. de Dios, J., Peña, R.: Formal certification of a resource-aware language implementation. In: TPHOLs '09, pp. 196–211. Springer, Heidelberg (2009)
6. Genet, T., Tong, V.V.T.: Timbuk 2.2., http://www.irisa.fr/celtique/genet/timbuk/
7. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Functional and Logic Programming (FLOPS 2010). LNCS. Springer, Heidelberg (2010)
8. Hardy, G.H., Ramanujan, S.: The normal number of prime factors of a number. Quart. J. of Math. 48, 76–92 (1917)
9. Java: The collections framework, http://java.sun.com/javase/6/docs/technotes/guides/collections/
10. Kaufmann, M., Moore, J.S.: An industrial strength theorem prover for a logic based on common lisp. IEEE Transactions on Software Engineering 23, 203–213 (1997)
11. Kuncak, V.: Binary search trees. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. Formal proof development (2004), http://afp.sf.net/entries/BinarySearchTree.shtml
12. Lammich, P.: Isabelle collection library. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. Formal proof development (2009), http://afp.sf.net/entries/collections.shtml
13. Lammich, P.: Tree automata. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. Formal proof development (2009), http://afp.sf.net/entries/Tree-Automata.shtml
14. LETHAL tree and hedge automata library, http://lethal.sourceforge.net/
15. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Nipkow, T., Pusch, C.: AVL trees. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. Formal proof development (2004), http://afp.sf.net/entries/AVL-Trees.shtml
17. Peyton Jones, S.: Bulk types with class. In: FPW '96 (1996)
18. Stepanov, A., Lee, M.: The standard template library. Technical Report 95-11(R.1), HP Laboratories (1995)

Interactive Termination Proofs Using Termination Cores*

Panagiotis Manolios and Daron Vroon

College of Computer and Information Science
Northeastern University
360 Huntington Ave., Boston MA 02115, USA
pete@ccs.neu.edu, daron.vroon@gmail.com

Abstract. Recent advances in termination analysis have yielded new methods and tools that are highly automatic. However, when they fail, even experts have difficulty understanding why and determining how to proceed. In this paper, we address the issue of building termination analysis engines that are both highly automatic and easy to use in an interactive setting. We consider the problem in the context of ACL2, which has a first-order, functional programming language. We introduce the notion of a *termination core*, a simplification of the program under consideration which consists of a single loop that the termination engine cannot handle. We show how to extend the Size Change Termination (SCT) algorithm so that it generates termination cores when it fails to prove termination, with no increase to its complexity. We show how to integrate this into the Calling Context Graph (CCG) termination analysis, a powerful SCT-based automatic termination analysis that is part of the ACL2 Sedan. We also present several new, convenient ways of allowing users to interface with the CCG analysis, in order to guide it to a termination proof.

1 Introduction

Recent years have seen great advances in the field of automated proofs of program termination (*e.g.* [1,3,8,11]). In this paper, we will explore one such termination analysis, the Calling Context Graph (CCG) algorithm, in detail [9,13]. The motivation for developing the CCG algorithm came from our desire to integrate mechanized program verification into the undergraduate curriculum. We used ACL2 [6,5,7], in part because it is based on a simple, applicative programming language and has a relatively simple logic. One of the first issues students confront is that functions must be shown to terminate. We wanted to avoid discussing ordinals and measure functions, so we developed and implemented the CCG termination analysis, which is able to automatically prove termination for the kinds of functions arising in undergraduate classes.

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 355–370, 2010.

Unfortunately, any termination analysis (and CCG is no exception) is bound to fail, either because the program under consideration is non-terminating or because the analysis is just not powerful enough to prove termination. In this paper, we address the issue of what to do when that happens. The idea is to leverage all of the information that was discovered during the termination effort. For example, the analysis may have discovered that certain loops in the program are terminating. Rather than throwing out all of that analysis and asking the user to prove termination from scratch, we want to present the user with an explanation of why the termination analysis failed. We propose to do that by generating a *termination core*, a new program that the termination analysis cannot prove terminating and which corresponds to a single simple cycle of the original program. Termination cores reveal the true reason that the termination analysis failed. Termination cores are a general notion that we believe can be fruitfully applied to any number of termination analyses, but in this paper we show how to compute termination cores for algorithms based on Size Change Termination (SCT) [8].

We start by reviewing CCG analysis in Section 2. We then introduce the notion of termination cores in Section 3. We prove that the complexity of the termination core generation problem for SCT is PSPACE-complete and we provide a practical algorithm. Just reporting termination cores is not enough. We need a mechanism that enables the user to interact with the termination analysis. We discuss this in Section 4, where we present several new, convenient ways of allowing users to interface with the CCG analysis, in order to guide it to a termination proof.

All the techniques described here are implemented in the current version of ACL2s, the ACL2 Sedan [4,2], a freely available, open-source, well-supported theorem prover that is based on ACL2, but was designed with greater usability and automation as primary design considerations. ACL2s provides a modern integrated development environment and includes fully automatic bug-finding methods based on a synergistic combination of theorem proving and random testing. CCG analysis is an integral part of ACL2s. In extensive experimental trials, it was able to prove over 98% of the more than 10,000 functions in the ACL2 regression suite terminating with no user input. ACL2s has been used at Northeastern University, UT Austin, and Georgia Tech to teach hundreds of undergraduate students how to reason about programs.

2 Termination Using Calling Context Graphs

We give a brief, simplified overview of the CCG analysis. For a more complete and detailed treatment, see [9,13]. The domain for the CCG analysis is a universe of programs, PROG, written in an applicative first-order functional programming language. For the sake of simplicity, we limit our discussion here to a very simple language, part of whose semantics is sketched in Figure 1: $\mathcal{F}$ denotes the universe of function names; $\mathcal{X}$ the universe of variable names; $\mathcal{V}$ the universe of values; $\mathcal{E}$ the universe of expressions; HIST the universe of *histories*, which map previously

$$f, g \in \mathcal{F} \qquad v, u \in \mathcal{V} \qquad x, y, z \in \mathcal{X} \qquad e, m \in \mathcal{E}$$
$$h \in \textsc{Hist} = \mathcal{F} \to \mathcal{X}^* \times \mathcal{E} \qquad \epsilon \in \textsc{Env} = \mathcal{X} \to \mathcal{V}$$

$$[\![\texttt{if } e_{test} \texttt{ then } e_{then} \texttt{ else } e_{else}]\!]^h\, \epsilon = \begin{cases} [\![e_{then}]\!]^h\, \epsilon \text{ if } [\![e_{test}]\!]^h\, \epsilon \neq \texttt{nil} \\ [\![e_{else}]\!]^h\, \epsilon \text{ otherwise} \end{cases}$$

$$[\![f\ e_1\ e_2\ \ldots\ e_n]\!]^h\, \epsilon = [\![e]\!]^h\, [x_i \mapsto v_i]_{i=1}^n \quad \text{where } v_i = [\![e_i]\!]^h\, \epsilon \text{ and } h(f) = \langle \langle x_i \rangle_{i=1}^n, e \rangle$$

Fig. 1. A rough sketch of a simple language and its semantics

defined functions to their signature and definitions; and Env the universe of environments, which map variables to values.

Consider the following function definition:

```
f x y = if (x < y) then 1 + (f (x+1) y)
        else if (x > y) then 1 + (f x (y+1))
        else 0
```

First, we create an abstraction of the program that captures its recursive behaviors while ignoring everything else. In `f`, the value returned in the base case and the fact that we add `1` to the value returned by each recursive call is irrelevant to the termination proof. We therefore reduce the program to its *calling contexts*. Intuitively, these are the recursive calls of the program along with the conditions under which each call is made. More formally, given a program $F \in$ Prog, a calling context is a triple, $\langle f, G, e \rangle \in \textsc{Contexts} = \mathcal{F} \times 2^{\mathcal{E}} \times \mathcal{E}$, such that f is a function defined in F (F can contain several function definitions), e is a call to a function defined in F, and G is the set of *governors* of e, *i.e.*, the *exact* set of conditions under which e is executed. The calling contexts for `f` are as follows.

1. $\langle$`f`, {`x < y`}, `(f (+ x 1) y)`$\rangle$
2. $\langle$`f`, {`not (x < y)`, `x > y`}, `(f x (+ y 1))`$\rangle$

Note that, in the governor for the second context, the second condition implies the first. We could therefore simplify this governor to be {`x > y`}.

The calling contexts are used to approximate the behavior of the program via the construction of a *Calling Context Graph (CCG)*, whose nodes are the calling contexts of the program and whose edges represent possible paths of execution from one context to the next. The minimal CCG for F is as follows.

Notice that if `x` $<$ `y`, then in the next iteration, `x` $\leq$ `y`, since `x` is incremented by `1`. Likewise, if `x` $>$ `y`, then in the next iteration, `x` $\geq$ `y` since `y` is incremented by `1`. Therefore, it is not possible for execution of `f` to move from one context to the other. This is a critical observation for proving termination, since if the flow of the program could alternate between the contexts, it could enter an infinite loop where `x` was increased, then `y`, and so on, without `x` and `y` ever being equal.

In order to formalize our notion of a CCG, we need the notion of a *call substitution*. Given a function call, $e = f\ \ e_1\ \ e_2\ \ \ldots\ \ e_n$, to a function with parameters $x_1, x_2, \ldots, x_n$, the call substitution for e, denoted σ_e, substitutes each e_i for the corresponding x_i.

A CCG is a graph, $\mathcal{G} = \langle C, E \rangle$, whose nodes, $C \subseteq$ CONTEXTS, and whose edges, $E \subseteq$ CONTEXTS $\times$ CONTEXTS and for any pair of contexts, $c_1 = \langle f_1, G_1, e_1 \rangle$, $c_2 = \langle f_2, G_2, e_2 \rangle \in C$, if e_1 is a call to f_2 and $\left[\!\left[\bigwedge_{g_1 \in G_1} g_1 \wedge \bigwedge_{g_2 \in G_2} g_2 \sigma_{e_1} \right]\!\right]^h \epsilon \neq \texttt{nil}$ for some $\epsilon \in$ ENV, then $\langle c_1, c_2 \rangle \in E$. Such an environment is called a *witness* for $\langle c_1, c_2 \rangle$. Notice that it is in general undecidable to determine if an edge must be in a CCG. This is why the condition for including an edge is an if rather than an iff. The goal is to create a safe approximation of the minimal CCG. Also, note that the *trivial CCG*, defined as the CCG in which there is an edge from c_1 to c_2 iff c_1 represents a call to the function containing c_2, gives us the exact same information as a standard call graph (in which the nodes are function names and there is an edge between f and g if f contains a call to g). We use theorem prover queries to generate CCGs that are smaller than the trivial one [9]. For example, using the ACL2 theorem prover, we are able to generate the minimal CCG for `f` as given above.

The next step in the CCG analysis is to annotate each edge of the CCG with a *generalized size change graph (GSCG)*. These tell us which values are decreasing or non-increasing from one context to the next in our CCG. A valid set of GSCGs for `f` is as follows.

G_1: $1 \to 1$: `y-x` $\xrightarrow{>}$ `y-x` $\qquad$ G_2 : $2 \to 2$: `x-y` $\xrightarrow{>}$ `x-y`

GSCGs are then used to annotate the CCG, creating an *Annotated CCG (ACCG)*. More formally, we define ACCGs and GSCGs as follows.

$$
\begin{aligned}
p, q, r \in \text{LAB} &= \{>, \geq\} \\
\mathcal{G}, \mathcal{H} \in \text{ACCG} &= 2^{\text{CONTEXTS}} \times 2^{\text{CONTEXTS} \times \text{GSCG} \times \text{CONTEXTS}} \\
G, H \in \text{GSCG} &= 2^{\mathcal{E}} \times 2^{\mathcal{E}} \times 2^{\mathcal{E} \times \text{LAB} \times \mathcal{E}} \times \text{CONTEXTS} \times \text{CONTEXTS}
\end{aligned}
$$

Each edge in the ACCG is annotated with a GSCG. We write $c_1 \xrightarrow{G} c_2$ to denote that $\langle c_1, G, c_2 \rangle$ is an edge $\in \mathcal{G}$. A GSCG is a bipartite graph with a set of expressions corresponding to the left nodes, a set of expressions corresponding to the right nodes, a set of labeled edges, and the pair of contexts the GSCG annotates. The tuple $\langle M_1, M_2, E, c_1, c_2 \rangle$ is a GSCG if for every $\langle m_1, r, m_2 \rangle \in E$ we have that $[\![m_1]\!]^h \epsilon \quad r \quad [\![m_2]\!]^h \epsilon$ for each ϵ that is a witness for $\langle c_1, c_2 \rangle$. We write $m_1 \xrightarrow{r} m_2$ to denote that $\langle m_1, r, m_2 \rangle$ is an edge $\in G$.

GSCGs and ACCGs are similar in concept to *size change graphs (SCGs)* and *annotated call graphs (ACGs)* that form the basis of the Size Change Termination analysis of Lee, Jones, and Ben-Amram for use in their size-change analysis [8]. The differences are that GSCGs have arbitrary expressions rather than just variables for nodes, and ACCGs mirror the recursive flow from recursive call to recursive call rather than from function to function. The result is a more

detailed analysis of program behavior. However, structurally, these concepts are the same, which allows us to apply the size change analysis to ACCGs as follows.

Definition 1. *A **multipath** π through an ACCG $\mathcal{G}$ is a (potentially infinite) path in $\mathcal{G}$: $\pi = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \xrightarrow{G_3} \cdots$.*

We write $\mathcal{G}^\omega$ for the set of infinite multipaths over $\mathcal{G}$ and $\mathcal{G}^+$ for the set of finite, nonempty ones. We sometimes write $G_1, G_2, \ldots$ or $\langle G_i \rangle$ to describe a multipath when the function names are irrelevant. Paths in ACCGs are called *multipaths* because their elements are graph structures and may contain many *threads*.

Definition 2. *A **thread** in a multipath $\pi = \langle G_i \rangle$ is a sequence of size-change edges $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ such that $x_{i-1} \xrightarrow{r_i} x_i \in G_i$ for all $i > 0$.*

For example, consider the multipath `f` $\xrightarrow{G_1}$ `f` $\xrightarrow{G_1}$ `f`. Its only thread is `y-x` $\xrightarrow{\geq}$ `y-x` $\xrightarrow{>}$ `y-x`. A thread tells us that the values of certain expressions do not increase during a sequence of calls, and can be used to prove termination as follows.

Definition 3. *The **Size Change Termination (SCT) problem** takes an ACCG as input and returns* `true` *if every infinite multipath through the ACCG has a suffix with a thread $\langle m_i \xrightarrow{r_i} m_{i+1} \rangle$ such that infinitely many $r_i = >$, or* `false` *otherwise.*

By well-foundedness, no infinite path through the ACCG that has such a thread can be an actual computation.

Theorem 1 (Lee, et al. [8]). *SCT is PSPACE complete.*

The SCT problem can be solved by *composing* GSCGs to create new GSCGs representing multiple transitions in the ACCG.

Definition 4. ***Composition** of GSCG labels and GSCGs is defined as follows.*

1. $p \cdot q = \begin{cases} \geq \text{ if } p = \geq \text{ and } q = \geq \\ > \text{ otherwise} \end{cases}$
2. $G_1 \cdot G_2 = \langle M_1, M_3, E, c_1, c_3 \rangle$, *where* $G_1 = \langle M_1, M_2, E_1, c_1, c_2 \rangle$, *and* $G_2 = \langle M_2, M_3, E_2, c_2, c_3 \rangle$, *and* $E = \{m_1 \xrightarrow{p \cdot q} m_3 \mid m_1 \xrightarrow{p} m_2 \in G_1 \ \wedge \ m_2 \xrightarrow{q} m_3 \in G_2\}$

Definition 5. *The **evaluation** of $\pi = \langle G_1, G_2, \ldots, G_n \rangle \in \mathcal{G}^+$ is $[\![\pi]\!] = G_1 \cdot G_2 \cdots G_n$.*

Proposition 1. *$m \xrightarrow{r} m' \in [\![\pi]\!]$ iff there exists a thread $m \xrightarrow{r_1} m_1 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} m_{n-1} \xrightarrow{r_n} m'$ in π, with $r = r_1 \cdot \ \cdots \ \cdot r_n$.*

Composition occurs until a fixed point is reached, at which point certain GSCGs, called *idempotents* are examined. An idempotent is a GSCG, G, of the form $\langle M, M, E, c, c \rangle$ such that $G \cdot G = G$. If all idempotents have an edge, $e \xrightarrow{>} e \in G$, then the algorithm returns `true`. Otherwise, it returns `false`.

Theorem 2 (Lee et al. [8]). *The algorithm above solves SCT.*

Theorem 3 (Manolios and Vroon [9]). *If a program, $D \in$ PROG has a corresponding ACCG, $\mathcal{G}$, such that SCT($\mathcal{G}$) = `true`, D is terminating on all inputs.*

3 Termination Cores

The idea behind termination cores is to present the user with a single simple cycle that embodies the reason for a failure to prove termination. We want this to be a general notion that applies to any termination prover, so we begin by defining a general notion of a termination analysis.

Definition 6. *A **termination analysis**, $\mathcal{T}$, is a function that takes in a set of function definitions, F, and returns* `true` *or* `false`*, such that if $\mathcal{T}(F) =$* `true`*, it is the case that the definitions of F will terminate for all inputs according to the semantics of the language.*

When the termination analysis fails, we want to create a new program that is simpler than the original, but still reflects the reason for the failure. We therefore must link the recursive behaviors of two programs, which we express as a relationship between their CCGs.

Definition 7. *Two calling contexts, c, c' of the form $\langle f, G, (g\ e_1\ \ldots\ e_n)\rangle$ and $\langle f', G, (g'\ e_1\ \ldots\ e_n)\rangle$, respectively, are said to be **similar**, denoted $c \sim c'$. We can denote c' as $[c]^{f'}_{g'}$, or c as $[c']^{f}_{g}$.*

Using this notion of context similarity, we develop the notion of a *similarity-preserving path homomorphism* between ACCGs that will form the basis of our definition of termination cores. We begin by defining path homomorphism.

Definition 8. *Given two directed graphs, $G = (C, E), G' = (C', E')$, a **path homomorphism** is a function $\phi : C \to C'$ such that $\langle c_1, c_2\rangle \in E \Rightarrow \langle \phi(c_1), \phi(c_2)\rangle \in E'$. If such a ϕ exists, we say that G is homomorphic to G'. If C and C' are sets of contexts, we say ϕ is **similarity-preserving** if $c \sim \phi(c)$ for all $c \in C$.*

In other words, a similarity-preserving homomorphism from $\mathcal{G}$ to $\mathcal{G}'$ demonstrates that the original program associated with $\mathcal{G}'$ contains a superset of the recursive behaviors of the new program associated with $\mathcal{G}$.

Definition 9. *Let $\mathcal{T}$ be a termination analysis, F be a set of function definitions such that $\mathcal{T}(F) =$* `false`*, and $\mathcal{G} = (C, E)$ be a CCG for F. Then a **termination core** for F modulo $\mathcal{T}$, is a set of function definitions, F' that satisfy all of the following:*

- *The trivial CCG, $\mathcal{G}' = (C', E')$, of F' is a simple cycle,*
- *There exists a similarity-preserving path homomorphism, $\phi : \mathcal{G}' \to \mathcal{G}$,*
- *$\mathcal{T}(F') =$* `false`*.*

Thus, the termination core is a single loop through the original program for which the termination analysis fails. The *Termination Core modulo $\mathcal{T}$ ($TC_{\mathcal{T}}$)* is the problem of finding such cores.

Definition 10. *Given a termination analysis $\mathcal{T}$, the* ***termination core modulo $\mathcal{T}$ ($TC_{\mathcal{T}}$) problem*** *takes a program P as input and generates* `null` *if $\mathcal{T}(P) =$* `true` *and a termination core for P modulo $\mathcal{T}$ otherwise.*

3.1 Termination Cores in CCG via Size Change Cores

In order to create termination cores for CCG, we use the notion of *size change cores (SCC).*

Definition 11. *A* ***size change core*** *is a finite multipath of the form $\pi = c \xrightarrow{G_1} c_1 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c$ such that, π^ω has no suffix with a corresponding thread of infinite descent.*

Proposition 2. *$SCT(\mathcal{G}) =$* `false` *iff there exists a size change core for $\mathcal{G}$.*

Proof. By the definition of SCT, it is clear that if such an SCC exists, $SCT(\mathcal{G}) =$ `false`. For the other direction, suppose $SCT(\mathcal{G}) =$ `false`. Then by Theorem 2, there exists $\pi = c \xrightarrow{G_1} c_1 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c$ such that $[\![\pi]\!]$ is idempotent and has no edge of the form $m \xrightarrow{>} m$. Notice that π is an SCC: by Proposition 1, there is no thread for π, $m \xrightarrow{r_1} m_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} m$ such that one of $r_1 \cdots r_n$ is "$>$". Now, suppose that π^ω has a suffix with an infinitely decreasing thread. By the pigeon hole principle, this means that there exists some k and m such that π^k has a thread $m \xrightarrow{r_1} m_1 \xrightarrow{r_2} \cdots \xrightarrow{r_{nk}} m$ such that some r_i is "$>$". By Proposition 1, this means that $[\![\pi^k]\!]$ has an edge $m \xrightarrow{>} m$ in it. But $[\![\pi^k]\!] = [\![\pi]\!]$ since $[\![\pi]\!]$ is idempotent. Therefore, no such edge can exist. □

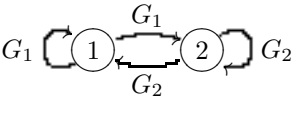
Consider the following function:

```
f a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
            then 1 + (f (b+1) (a-1) (c+1) (d-1))
                   + (f (b-1) (a+1) (c-1) (d+1))
            else 0
```

The contexts for this function are as follows.

1. ⟨`f`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f (b+1) (a-1) (c+1) (d-1))`⟩
2. ⟨`f`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f (b-1) (a+1) (c-1) (d+1))`⟩

Suppose we use the measures `a`, `b`, `c`, and `d` for both contexts. Then we get the following ACCG.



Notice, then, that an SCC for this program is $1 \xrightarrow{G_1} 1 \xrightarrow{G_1} 2 \xrightarrow{G_2} 1$, for which there is no thread at all. The problem is that this is not a simple cycle. In order to create a termination core, then, we must derive a simple cycle that corresponds to an SCC. We do this by renaming the functions in the contexts in order to distinguish between occurrences of a single context in the loop.

Definition 12. *Given a cycle $\boldsymbol{c} = c_1, c_2, \ldots, c_n, c_1$ in an ACCG, and a sequence of fresh, distinct function symbols, $\boldsymbol{f} = \langle f_i \rangle_{i=1}^{n} \in \mathcal{F}^n$, the **similar simple cycle with respect to $\boldsymbol{f}$**, denoted $[\boldsymbol{c}]_{\boldsymbol{f}}$, is the sequence $\boldsymbol{c'} = \langle [c_i]_{f_{i+1}}^{f_i} \rangle$ where $f_{n+1} = f_1$. Given a cyclical multipath, $\pi = c_1 \xrightarrow{G_1} c_2 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c_1$, the **set of similar simple cyclical multipaths with respect to $\boldsymbol{f}$**, denoted $[\pi]_{\boldsymbol{f}}$ is the set containing all $\pi' = [c_1]_{f_2}^{f_1} \xrightarrow{G'_1} [c_2]_{f_3}^{f_2} \xrightarrow{G'_2} \cdots \xrightarrow{G'_n} [c_1]_{f_2}^{f_1}$ such that each G'_i is a subgraph of the corresponding G_i.*

The two key features of similar simple cyclical multipaths are that they are simple cycles and that they preserve the non-terminating behavior of an SCC. Thus, we get the following result.

Lemma 1. *Let $\pi = \langle G_i \rangle_{i=1}^{n}$ be an SCC and $\pi' = \langle G'_i \rangle_{i=1}^{n} \in [\pi]_{\boldsymbol{f}}$. Then π' is an SCC of the ACCG consisting of the contexts, edges, and GSCCs of π'.*

Proof. By definition, there is no infinitely decreasing thread corresponding to π^ω. But every thread of π^ω is a thread of π'^ω by construction. Therefore, there is no infinitely decreasing thread corresponding to π'^ω. By definition, this makes π' an SCC.

Returning to our example, we can construct a similar simple cycle to our SCC as follows:

1. $\langle$`f0`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f1 (b+1) (a-1) (c+1) (d-1))`$\rangle$
2. $\langle$`f1`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f2 (b+1) (a-1) (c+1) (d-1))`$\rangle$
3. $\langle$`f2`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f0 (b-1) (a+1) (c-1) (d+1))`$\rangle$

We use a similar simple cycle to an SCC to create a corresponding termination core. We do this by creating a trivial function for each context in the similar simple cycle as follows.

Definition 13. *Given a calling context, $c = \langle f, \{e_1, e_2, \ldots, e_n\}, e \rangle$, the **minimal function definition for** c is the following function, where $\langle x_i \rangle_{i=1}^{m}$ are the parameters for function f.*[1]

$f\ x_1\ x_2\ \ldots\ x_m$ `=`
  `if (`e_1 `and` e_2 `and ... and` e_n`) then` e `else`
  `[`x_1`;` x_2`; ...;` x_m`]`

The point is to use the similar simple cycles to construct the minimal definitions. In the case of our example, we have the following.

[1] Notice that we can return anything at all in the `else` case below. In our implementation, we return the list of parameters for technical, ACL2-related reasons.

```
f0 a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
             then (f1 (b+1) (a-1) (c+1) (d-1))
             else [a; b; c; d]
f1 a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
             then (f2 (b+1) (a-1) (c+1) (d-1))
             else [a; b; c; d]
f2 a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
             then (f0 (b-1) (a+1) (c-1) (d+1))
             else [a; b; c; d]
```

What remains is to prove that this corresponds to a termination core in general. Our work thus far allows us to prove that the resulting functions satisfy the first two conditions of termination cores.

Lemma 2. *Let* $\pi = c_1 \xrightarrow{G_1} c_2 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c_1$ *be a size change core for ACCG* $\mathcal{G}$, $\boldsymbol{c} = c_1, c_2, \ldots, c_n, c_1$, $\boldsymbol{f}$ *be a sequence of fresh, distinct function names, and* $\boldsymbol{c'} = [\boldsymbol{c}]_{\boldsymbol{f}} = c'_1, c'_2, \ldots, c'_n, c'_1$. *Then the trivial CCG,* $\mathcal{G}'$ *of the minimal function definitions of* $\boldsymbol{c'}$, *is a simple cycle such that there exists a similarity-preserving path homomorphism from* $\mathcal{G}'$ *to* $\mathcal{G}$.

Proof. This follows from the freshness and distinctness of the function names in $\boldsymbol{f}$. □

All that remains, then, is to show that our construction results in functions that cannot be proved terminating by our analysis. In order to do this, we need to make some assumptions about the construction of ACCGs. Intuitively, we need to assume that we are consistent in our choice of measures and our ability to prove the necessary queries. We require that our ACCG generator is not "smarter" when analyzing the termination core than it is when analyzing the the original function.

Definition 14. *Let* `build-accg` : PROG → *ACCG be a function that computes an ACCG corresponding to the input program. Then we say that* `build-accg` *is* ***monotonic*** *if when given* $P, P' \in$ PROG *such that there exists a similarity-preserving homomorphism,* ϕ *from the trivial CCG of* P *to the trivial CCG of* P', *the following conditions hold, where* $\mathcal{G}_P =$ `build-accg`(P) *and* $\mathcal{G}_{P'} =$ `build-accg`(P'):

- $c_1 \xrightarrow{G} c_2 \in \mathcal{G}_P$ *if the call of* c_1 *is a call to the function containing* c_2 *and* $\phi(c_1) \xrightarrow{G'} \phi(c_2) \in \mathcal{G}_{P'}$, *and*
- *For all* $c_1 \xrightarrow{G} c_2 \in \mathcal{G}_P$ *and* $\phi(c_1) \xrightarrow{G'} \phi(c_2)$, G *is a subgraph of* G'.

From this point forward, we assume a fixed, monotonic `build-accg`. The interesting about this monotonicity property for us is that it means that the ACCG we construct for our termination core contains a similar simple multipath to the SCC we used to construct it. More formally, we have the following.

Lemma 3. *Let P be a program and $\mathcal{G}_P = \texttt{build-accg}(P)$ such that π is an SCC for $\mathcal{G}_P$ that traverses contexts $\boldsymbol{c}$. Then if $\boldsymbol{c'} = [\boldsymbol{c}]_{\boldsymbol{f}}$ and P' is the set of minimal function definitions for $\boldsymbol{c'}$, then the multipath π' that visits in order the contexts of $\boldsymbol{c'}$ in $\mathcal{G}_{P'} = \texttt{build-accg}(P')$ is a similar simple multipath to π.*

Proof. Follows from the definition of monotonicity. □

Now we fix our definition of `ccg` and `ccg-tc`, our termination analysis and termination core solver, respectively; but first, we introduce the size change core problem.

Definition 15. *The **Size Change Core (SCC) problem** takes an ACCG, $\mathcal{G}$, as input and returns a size change core if SCT($\mathcal{G}$) =* `false` *and* `null` *otherwise.*

Definition 16. *We define* `ccg` *as SCT*(`build-accg`(P)). *We define* `ccg-tc` : Prog → Prog *as follows: Given program, P,let $\pi = SCC(\texttt{build-accg}(P))$. If $\pi =$* `null`*, return* `null`*. Otherwise, return the minimal function definitions for $[\pi]_{\boldsymbol{f}}$ for some fresh function names, $\boldsymbol{f}$.*

Based on the work we've done so far, it is fairly straightforward to prove the following.

Theorem 4. `ccg-tc` *solves $TC_{\texttt{ccg}}$.*

Proof. By the definition of SCC, `ccg-tc`(P) returns `null` iff `ccg`$(P) =$ `true`, so we only need to concern ourselves with the case in which `ccg`$(P) =$ `false`.

If `ccg-tc`(P) produces a program' P', then P' satisfies the first two properties of a termination core by Lemma 2. It satisfies the final property by a combination of Lemma 3 with Lemma 1. □

3.2 Constructing Size Change Cores in CCG

We prove that $SCC \in PSPACE$. To do this, we use a characterization of the problem using Büchi automata. Given an ACCG, $\mathcal{G}$, consider the two sets of infinite multipaths:

$$FLOW^\omega = \mathcal{G}^\omega$$
$$DESC^\omega = \{\pi \in \mathcal{G}^\omega \mid \pi \text{ has a suffix with an infinitely decreasing thread}\}$$

By results in [8], both $FLOW^\omega$ and $DESC^\omega$ are ω-regular subsets of GSCG$^\omega$ for which there are Büchi automata that solve each in space polynomial with respect to the size of the original program. Note that SCT is equivalent to determining if $FLOW^\omega \subseteq DESC^\omega$. This, in turn can be expressed as the problem of determining that $FLOW^\omega \cap \overline{DESC^\omega}$ is empty.

What we want is to find $\pi \in \mathcal{A} = FLOW^\omega \cap \overline{DESC^\omega}$ when such a π exists. One idea is to construct the Büchi automaton corresponding to $\mathcal{A}$ and to search for such a path. Unfortunately, this does not work because complementing a Büchi automaton can lead to an exponential blowup. Fortunately, there are methods that allow us to traverse $\mathcal{A}$ in polynomial space without actually constructing it [10]. The PSPACE completeness of SCC can then be proved as follows.

Theorem 5. *SCC is PSPACE complete.*

Proof. Showing PSPACE hardness is trivial. By Proposition 2, SCT is reducible to SCC, and by Theorem 1, SCT is PSPACE-complete.

To show that SCC is in PSPACE, we non-deterministically find a multipath $\pi = \pi_1\pi_2 \in \mathcal{A}$ such that $s \xrightarrow{\pi_1} a \xrightarrow{\pi_2} a$ for some initial state s and accepting state a, using the following algorithm, where S^0 is the set of initial states, S is the set of states, and F is the set of final states of $\mathcal{A}$. Also, the alphabet of $\mathcal{A}$ is the set of GSCG's of the ACCG $\mathcal{G}$.

```
1:  s ← s_0 ← choose(S^0); a ← choose(F)
2:  for i = 1 to |S| do
3:      s ← choose({t ∈ S | s -G-> t ∈ A})
4:      if s = a then
5:          for i = 1 to |S| do
6:              G ← choose(G)
7:              s ← choose({t ∈ S | s -G-> t ∈ A})
8:              Output G
9:              if s = a then
10:                 return found
11:         return null
12: return null
```

Here, `choose` denotes a non-deterministic choice of an element in the given set if such an element exists. If not, it causes the entire algorithm to halt and return `null`. Note that $\mathcal{A}$ is non-empty iff such a path exists [12]. However, the algorithm only outputs π_2. By the definition of $\mathcal{A}$, we see that $\pi_1\pi_2^\omega$ is an infinite multipath such that no suffix of the multipath has an infinitely decreasing thread. Thus, π_2^ω is also such a multipath, since any suffix of π_2^ω is a suffix of $\pi_1\pi_2^\omega$. By definition, this makes π_2^ω a size change core. Therefore, this algorithm solves SCC.

At any given point, all we are storing is four states (s_0, a, s, and t), two counters between 1 and $|S|$, and a single GSCG, G. All of this plus determining if $s \xrightarrow{G} t \in \mathcal{A}$ can be done in polynomial space. Therefore, $SCC \in NPSPACE = PSPACE$. □

Definition 17. *An **enhanced size change graph (ESCG)** is a triple, $\langle G, p, l\rangle$ where G is a GSCG and one of the two conditions hold:*

- *$p = G$ is an GSCG and $l = 1$, or*
- *p is a pair of ESCGs, $\langle H', H''\rangle$ such that $G = G' \cdot G''$, and $l = l' + l''$, where $H' = \langle G', p', l'\rangle$ and $H'' = \langle G'', p'', l''\rangle$.*

*The **enhancement** of an existing GSCG, G, denoted $\overline{G}$, is the ESCG, $\langle G, G, 1\rangle$.*

Definition 18. *The **corresponding multipath** of an ESCG, $H = \langle G, p, l\rangle$, denoted $\mathtt{path}(H)$, is G if $p = G$, and $\mathtt{path}(H_1)\mathtt{path}(H_2)$ if $p = \langle H_1, H_2\rangle$.*

Definition 19. *The **composition** of two ESCGs, $H_1 = \langle G_1, p_1, l_1\rangle$ and $H_2 = \langle G_2, p_2, l_2\rangle$, denoted $H_1 \cdot H_2$ is the ESCG, $\langle G_1 \cdot G_2, \langle H_1, H_2\rangle, l_1 + l_2\rangle$.*

Let $S' \leftarrow$ the enhancements of all the GSCGs of the ACCG.
Let $\prec$ s.t. $\langle G, p, l\rangle \prec \langle G', p', l'\rangle$ iff $l < l'$.
repeat
  $S \leftarrow S'$
  for all $H = \langle G, p, l\rangle, H' = \langle G', p', l'\rangle \in S$ **do**
    if $\exists H'' = \langle G'', p'', l''\rangle \in S'$ s.t. $G'' = G \cdot G'$ **then**
      $S' \leftarrow S' - \{H''\} \cup \min_{\prec}\{H \cdot H', H''\}$
    else
      $S' \leftarrow S' \cup \{H; H'\}$
until $S = S'$
if $\exists H = \langle G, p, l\rangle$ s.t. G is idempotent with no edge of the form $m \xrightarrow{>} m$ **then**
  return $\mathtt{path}(H)$
else
  return `true`

Fig. 2. An algorithm for solving SCC

Lemma 4. *Given an ESCG,* $H = \langle G, p, l\rangle$, $G = [\![\mathtt{path}(H)]\!]$.

Theorem 6. *The algorithm given in Figure 2 solves SCC.*

Proof. The algorithm behaves exactly as the one in Theorem 2, except that we keep track of which ESCGs were composed to create each new ESCG and the length of the path corresponding to the ESCG. Thus there exists an idempotent ESCG without an edge $m \xrightarrow{>} m$ iff SCT also discovers such an edge. Therefore, if $SCT(\mathcal{G}) = \mathtt{true}$, $SCC(\mathcal{G}) = \mathtt{null}$, and if $SCT(\mathcal{G}) = \mathtt{false}$, $SCC(\mathcal{G})$ returns a circular multipath, π such that $[\![\pi]\!]$ is idempotent and contains no edge of the form $m \xrightarrow{>} m$. Suppose there was some decreasing thread corresponding to π^ω. Then by the pigeon hole principle, there would be some k such that π^k such that there existed a decreasing thread from some m to itself. By Proposition 1, this means that $[\![\pi^k]\!]$ would have an edge $m \xrightarrow{>} m$. But by the definition of idempotence, $[\![\pi^k]\!] = [\![\pi]\!]$, and we already stated that no $[\![\pi]\!]$ has no edge of the form $m \xrightarrow{>} m$. Therefore, there is no infinite decreasing thread for π^ω. By definition, then, π is an SCC. □

Theorem 7. *The algorithms in Theorem 2 and Figure 2 have the same complexity.*

Proof. To the original data structures, we add pointers to two ESCGs and an integer representing the length of a path whose evaluation leads to the corresponding ESCG. Since there can be exponential ESCGs, these added fields require linear length in the size of the original problem. In the loop itself, we perform one addition and one comparison of the lengths, which takes linear time, and create two pointers to the ESCGs being composed, which takes constant time. This is eclipsed by the composition of the GSCGs, which has complexity that is polynomial and greater than linear. Thus, there is no overall change in the complexity from the SCT algorithm to the SCC algorithm. □

4 Interactive CCG

Termination core analysis, as described in this paper is implemented in ACL2s, the ACL2 Sedan. When reporting termination cores to ACL2s users, our goal is to give as specific and concrete a reason for the failure to prove termination as possible, taking into account everything the termination prover discovered in its proof attempt. The hope is that termination cores will be effective tools for helping users efficiently debug failed termination proofs.

Once a user figures out why the termination proof attempt failed, she must then be able to interact with the termination engine, in order to guide it towards a proof. We have developed a clean and intuitive interface that gives the user a way to interact with the theorem prover without getting bogged down in the details of CCG analysis. The interface is based on the three possible reasons CCG can fail to prove termination.

The first and most obvious source of failure is a non-terminating program. In this case, our analysis will reach a point at which it finds an SCC that represents an actual infinite run of the program. This is particularly useful in programs with multiple recursive behaviors, as it enables the user to find the relevant code and make revisions. Consider, for example, the following program. The reader is encouraged to figure out what is going on before reading further.

```
f1 w r z s x y a b zs =
  if (a > 0) then
     f2 w r z 0 r w 0 0 zs
  else w = r^zs

f2 w r z s x y a b zs =
  if z > 0 then
     f3 w r z s x y y s zs
  else f1 s r (z-1) 0 0 0 0 0 zs

f3 w r z s x y a b zs =
   if a > 0 then
      f3 w r z s x y (a-1) (b+1) zs
   else f2 w r z b (x-1) y 0 0 zs
```

ACL2s produces the following core:

```
f3_0 w r z s x y a b zs =
  if a <= 0 then
    f2_0 w r z b (x-1) y 0 0 zs
  else
    [w; r; z; s; x; y; a; b; zs]

f2_0 w r z s x y a b zs
  if z > 0 then
    f3_0 w r z s x y y s zs
  else
    [w; r; z; s; x; y; a; b; zs]
```

From the termination core, we see that the only value consistently decreasing in this loop is `x`, which decreases by 1 each time through the loop. The problem is that there is no test to see that `x` is positive. Instead, we check that `z` is positive. This is easily remedied by changing `z > 0` to `x > 0` in the definition of `f2`. Does the program terminate now? Readers are encouraged to construct a measure and to mechanically verify it. (It took us about 20 minutes.) If we submit the updated program, CCG analysis proves termination in under 2 seconds, fully automatically with no user guidance.

The above program was generated by applying weakest precondition analysis to a triply-nested loop. An expert with over a decade of theorem proving

experience spent 4–6 hours attempting to construct a measure function that could be used to prove termination, before giving up. So, this example highlights how useful termination analysis can be and how termination core analysis can help users discover and correct termination bugs.

A second reason that CCG analysis may fail to prove termination is that it may fail to guess the necessary Calling Context Measures (CCMs) [9,13]. Calling context measures can be thought of as building blocks for conventional measures. We use simple heuristics to guess CCMs, and while they are effective for many programs, they are certainly not complete. Consider, the following program.

```
dec x = if x <= 0 then 255 else x-1
f x = if x = 1 then 0 else 1 + (f (dec x))
```

Our heuristics choose |x| as the CCM for the sole recursive call in `f`. However, this measure does not always decrease across the recursive call. For example, if `x` is `0`, `(dec x)` is `255`. The reader is encouraged to prove termination using the standard measure-based approach.

The termination core produced by our algorithm is as follows.

```
f_0 x = if x = 1 then [x] else (f_0 (dec x))
```

This termination core is not terribly helpful, since it has the exact same looping behavior as the original. However, our core generator also lists the CCMs chosen for each context, as well as the edges of the relevant GSCGs. For our example, our analysis will inform the user that the sole CCM chosen was |x|, which cannot be shown to be non-increasing or decreasing from one iteration of the loop to the next. A quick look at the definition of `dec` confirms that this is not an appropriate CCM. However, `(dec x)` is a useful CCM. That is, it is easy to prove that if `x` is not `1`, `(dec (dec x)) < (dec x)`.

We provide an interface that allows users to override the heuristics for guessing CCMs by providing one of two hints to the CCG algorithm. The first is the `:CONSIDER` hint, which takes a list of expressions over the parameters of the function to which the user wishes to apply the hint. This tells the CCG analysis to add the given CCMs to those heuristically generated for all the contexts in the function to which it is applied. In our example, a `:CONSIDER [(dec x)]` hint will result in measures {|x|, `(dec x)`} for the sole context.

In some cases, users may want even more control. For example, |x| is irrelevant for the termination proof. Such CCMs lead to needless theorem prover queries. Therefore, we also provide a `:CONSIDER-ONLY` hint. This is identical in usage to the `:CONSIDER` hint, but tells the CCG analysis to use *only* those measures provided by the user for the given function. Thus, the hint `:CONSIDER-ONLY [(dec x)]` in our example will result in `(dec x)` being the only measure for the sole context. Giving this hint leads to a simpler termination proof.

The final reason that CCG may fail to prove termination is that it was unable to prove a necessary theorem about either the exclusion of an edge from the CCG or about the relationship between two measures across a recursive call. Consider, for example the following definition of merge sort.

```
mergesort x =
   if x = [] or tl x = []
   then x
   else mergelists (mergesort (evens x)) (mergesort (odds x))
```

Here, `(evens` $[e_0;\ e_1;\ \ldots;\ e_n]$`)` returns $[e_0;\ e_2;\ \ldots]$ and `odds` applied to the same list returns $[e_1;\ e_3;\ \ldots]$. Our analysis produces the following core:

```
mergesort_0 x = if not (x = []) and (not (tl x = []))
                then  mergesort_0 (evens x) else [x]
```

It also tells us that the sole measure |`x`| (*i.e.*, the length of `x`) could not be shown to be decreasing. The problem is that the theorem prover was unable to prove that `evens` always returns a list that is smaller than the input if that input list has 2 or more elements in it. It turns out that ACL2d needs some guidance to get this proof to go through. If we prove this lemma, termination still fails, but we get a new core.

```
mergesort_0 x = if not (x = []) and (not (tl x = []))
                then mergesort_0 (odds x) else [x]
```

We have the same problem with `odds` that we had with `evens`. A similar lemma leads to a successful termination proof. Note the interactive nature of this process. There were two different reasons for CCGs inability to prove termination. Rather than simply giving up, the CCG analysis shows the user each reason for non-termination, one at a time, thereby enabling the user to successfully address these issues and prove termination.

5 Conclusions

We examined the issue of building termination analysis engines that are both highly automatic and easy to use in an interactive setting. The challenge is in understanding and then dealing with failure. To this end, we introduced the notion of a *termination core*, a simplification of the program under consideration which consists of a single loop that the termination engine cannot prove terminating. Termination cores are used to help users understand why a termination analysis engine failed to prove termination. We showed how to extend Size Change Termination so that it generates a termination core when it fails to prove termination. We showed that this is a PSPACE-complete problem and presented a practical algorithm that adds termination core generation to the Calling Context Graph termination analysis, a recent, powerful termination analysis that is part of the ACL2 Sedan. We also presented several convenient ways of allowing users to interact with CCG analysis, so that they can guide it to a termination proof after analyzing the termination core that was generated. These techniques are implemented in the ACL2 Sedan, a freely available, open-source, well-supported theorem prover that is based on ACL2, but was designed with greater usability and automation as primary design considerations.

References

1. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: POPL '08, pp. 101–112. ACM, New York (2008)
2. Chamarthi, H.R., Dillinger, P.C., Manolios, P., Vroon, D.: ACL2 Sedan homepage, http://acl2s.ccs.neu.edu/
3. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI '06, pp. 415–426. ACM, New York (2006)
4. Dillinger, P.C., Manolios, P., Vroon, D., Strother Moore, J.: ACL2s: "The ACL2 Sedan". Electr. Notes Theor. Comput. Sci. 174(2), 3–18 (2007)
5. Kaufmann, M., Manolios, P., Strother Moore, J. (eds.): Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, Dordrecht (June 2000)
6. Kaufmann, M., Manolios, P., Strother Moore, J.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Dordrecht (July 2000)
7. Kaufmann, M., Strother Moore, J.: ACL2 homepage, http://www.cs.utexas.edu/users/moore/acl2 (August 2007)
8. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL '01, vol. 28, pp. 81–92. ACM, New York (2001)
9. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 401–414. Springer, Heidelberg (2006)
10. Prasad Sistla, A., Vardi, M.Y., Wolper, P.: The complementation problem for büchi automata with applications to temporal logic. Theor. Comput. Sci. 49(2-3), 217–237 (1987)
11. Thiemann, R., Giesl, J.: Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen (January 2003)
12. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Banff Higher order workshop conference on Logics for concurrency, pp. 238–266. Springer, Heidelberg (1996)
13. Vroon, D.: Automatically Proving the Termination of Functional Programs. PhD thesis, Georgia Intitute of Technology (2007)

A Framework for Formal Verification of Compiler Optimizations

William Mansky and Elsa Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign,
Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302
{mansky1,egunter}@cs.illinois.edu

Abstract. In this article, we describe a framework for formally verifying the correctness of compiler optimizations. We begin by giving formal semantics to a variation of the TRANS language [6], which is designed to express optimizations as transformations on control-flow graphs using temporal logic side conditions. We then formalize the idea of correctness of a TRANS optimization, and prove general lemmas about correctness that can form the basis of a proof of correctness for a particular optimization. We present an implementation of the framework in Isabelle, and as a proof of concept, demonstrate a proof of correctness of an algorithm for converting programs into static single assignment form.

Keywords: optimizing compilers, theorem proving, program transformations, temporal logic.

1 Introduction

Optimizations for time and memory efficiency are now an essential feature of almost all modern compilers. These optimizations are often complex program transformations, and establishing their correctness is a difficult process. In this paper, we propose a general framework for expressing and verifying compiler optimizations using established theorem-proving tools, with the goal of reducing the burden of proving correct any particular optimization.

The problem of verifying a compiler optimization can be divided into two main parts. The first is *specification* of the optimization by giving its semantics in some mathematical formalism. For this purpose, we use the TRANS language, proposed by Kalvala et al. [6], in which optimizations are defined as transformations on control flow graphs conditioned on the satisfaction of temporal logic formulae. This approach is particularly amenable to proofs of correctness for several reasons. First, it allows a more modular formulation of many optimizations than the traditional algorithms, reducing the amount of context that needs to be drawn in to the proof of each step of optimization. Second, it provides a uniform framework for expressing various types of optimizations; by proving certain facts about the fundamental operations of the language, we can take advantage of the redundancy among different optimizations. Finally, the use of temporal logic side conditions makes the assumptions of the transformation explicit in a formal

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 371–386, 2010.

sense, and can help narrow the gap between the theoretical semantics of the optimization and its actual implementation (e.g., by using model checking to verify the condition before the transformation is performed). As part of the framework, we provide an Isabelle implementation of TRANS, and an instantiation capable of expressing the static single assignment (SSA) transformation.

The second part of the problem, and often the more intractable one, is *verification* of the optimization. Given a definition of correctness, one must construct a formal proof that the correctness property holds for the optimization (see for instance Leroy [8]). The complexity of this process can vary significantly depending on the choice of formalisms in previous steps. Optimizations are most commonly specified as algorithms operating on program code, an approach that is easy to implement but relatively difficult to verify. TRANS itself does not provide a verification procedure, but it does break optimizations down into combinations of simple graph transformations, offering a cleaner, more modular approach. By identifying the basic operations involved in common optimizations, and proving useful lemmas about their correctness, we hope to reduce the amount of effort involved in the proof of correctness of any particular optimization.

To demonstrate the versatility of the framework, we have defined and verified an algorithm for transforming programs into SSA form, a common precursor to compiler optimizations [3,1]. The SSA transformation is particularly interesting because it extends the underlying language of a program, adding ϕ-functions that are used to determine which instance of a variable should be used based on the program's execution trace. While the transformation cannot be expressed in TRANS as originally presented, we offer a parametric view of TRANS which allows us to easily add transformation-specific constructs such as indexed variables and ϕ-functions. We then give a formal proof of correctness for the SSA conversion using the Isabelle/HOL theorem prover, using the lemmas provided by the framework. The result is a specification of the SSA transformation that is guaranteed to preserve the semantics of the original program.

2 Language Framework

TRANS is a language for expressing program transformations in terms of conditional rewrites on control-flow graphs. As such, in order to give its formal semantics, we must first define the language of programs and formalize the notion of control-flow graphs for those programs. We begin with the simple language L_0, which captures some of the basic functionality of intermediate representations. An L_0 program is a list of instructions, and the number provided to a `goto` command gives the target instruction as a position in the list. More formally:

$$
\begin{aligned}
\mathit{expr} ::= {} & \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid \mathit{num} \mid \mathit{var} \\
\mathit{instr} ::= {} & \mathit{var} := \mathit{expr} \mid \texttt{ret}(\mathit{expr}) \mid \texttt{if}\ \mathit{expr}\ \texttt{goto}\ \mathit{num} \mid \texttt{skip} \mid \texttt{goto}\ \mathit{num} \\
\mathit{graph\_instr} ::= {} & \mathit{var} := \mathit{expr} \mid \texttt{ret}(\mathit{expr}) \mid \texttt{if}\ \mathit{expr} \mid \texttt{goto}
\end{aligned}
$$

The semantics of L_0 have been formalized by Lacey et al. [7].

Since TRANS operates on control-flow graphs (CFGs), rather than directly on programs, we must also formally define CFGs. Our definition is adapted from

that given by Kalvala [6], in which the nodes are labeled with single instructions rather than larger basic blocks. A CFG is a record $\mathcal{G} = (N, E \subseteq N \times N \times \{seq, branch\}, I : N \rightharpoonup graph\_instr$[1]$, S : N \rightharpoonup N)$, where $A \rightharpoonup B$ indicates a partial function from A to B, satisfying the following properties:

1. the node set $N_\mathcal{G}$ contains the special nodes *Entry* and *Exit*
2. the instruction labeling $I_\mathcal{G}$ is defined on all nodes except *Entry* and *Exit*
3. *Entry* has no incoming edges, and *Exit* has no outgoing edges
4. the outgoing edges of each other node are consistent with the instruction labeling of that node
5. the reflexive transitive closure of the successor relation $S_\mathcal{G}$ is a linear order
6. if two nodes m and n are connected by an edge with label *seq*, then n is the successor in $S_\mathcal{G}$ of m

Of these, property 4 captures the core idea of a CFG, and its formal definition depends on the underlying language of the graph. In the case of L_0, the appropriate edges for each instruction type are as follows:

- The *Entry* node has one outgoing edge, with label *seq*
- a node labeled with := has one outgoing edge, with label *seq*
- a node labeled with `if` has two outgoing edges, one with label *seq* and one with label *branch*
- a node labeled with `goto` has one outgoing edge, with label *branch*
- a node labeled with `ret` has one outgoing edge, with label *seq*, connecting to the *Exit* node

The inclusion of a successor function aids us in transforming CFGs back into programs, so that TRANS transformations can be said to operate on programs as well as graphs.

A useful property in establishing program correctness is *recoverability*. A CFG is recoverable if it has a unique last node: that is, there is only one node that is labeled with `ret`, and that node is the only predecessor of the *Exit* node[2] [6]. Reasoning about the correctness of a transformation can sometimes be simplified by adding the assumption that the graph in question is recoverable.

Since the framework is to be used to prove that certain transformations preserve the semantics of CFGs, it must also include a notion of evaluation of a CFG. We can give L_0 CFGs a small-step execution semantics, based on the semantics of L_0, in a manner similar to Leroy [8]. The configurations of a CFG under execution are either *intermediate configurations* of the form (m, l, t), where m is a memory, l is a node in the graph (a program point), and t is an execution trace[3], or *values* v, indicating that the execution of the graph has terminated.

[1] Rather than retain the program's instruction numbering in the graph, we use edges to indicate the targets of `goto` and `if` statements, which also erases the distinction between `goto` and `skip` nodes; we use `goto` for either sort of instruction.

[2] Note that any program with at least one `ret` instruction can be restructured to satisfy this condition, by replacing returns with jumps to a single return instruction.

[3] While the execution trace t does not affect the outcome of any instruction in L_0, we include it for generality; in particular, we will make use of it in adjusting the framework to handle the SSA transformation.

Then we can define the small-step relation $\rightarrow_{\mathcal{G}}$ for a graph $\mathcal{G}$ as follows, where we use the notation *out_edges* $\mathcal{G}$ l to indicate the set of outgoing edges of l in $\mathcal{G}$, and assume standard evaluation semantics for arithmetic expressions:

$$\frac{\mathit{out\_edges}\ \mathcal{G}\ \mathit{Entry} = \{(\mathit{Entry}, l', \mathit{seq})\}}{(m, \mathit{Entry}, t) \rightarrow_{\mathcal{G}} (m, l', \mathit{Entry}; t)}$$

$$\frac{I_{\mathcal{G}}(l) = x := e \qquad (e, m) \Downarrow v \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{seq})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m[x \leftarrow v], l', l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{if}\ e \qquad (e, m) \Downarrow 0 \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l'_1, \mathit{seq}), (l, l'_2, \mathit{branch})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l'_1, l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{if}\ e \quad (e, m) \Downarrow v \quad v \neq 0 \quad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l'_1, \mathit{seq}), (l, l'_2, \mathit{branch})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l'_2, l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{goto} \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{branch})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l', l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = \texttt{ret}(e) \qquad (e, m) \Downarrow v}{(m, l, t) \rightarrow_{\mathcal{G}} v}$$

We can now define precisely what we mean when we say that two CFGs are semantically equivalent. We define the set of results of a CFG $\mathcal{G}$ starting from a configuration (m, l, t) as the set of values in the transitive closure of the small-step relation:

$$\frac{(m, l, t) \rightarrow_{\mathcal{G}} v}{(m, l, t, v) \in \mathit{result}\ \mathcal{G}}$$

$$\frac{(m, l, t) \rightarrow (m', l', l; t) \qquad (m', l', l; t, v) \in \mathit{result}\ \mathcal{G}}{(m, l, t, v) \in \mathit{result}\ \mathcal{G}}$$

Then two graphs $\mathcal{G}$ and $\mathcal{G}'$ are semantically equivalent if and only if

$$\forall v.(\mathit{empty}, \mathit{Entry}, [], v) \in \mathit{result}\ \mathcal{G} \Leftrightarrow (\mathit{empty}, \mathit{Entry}, [], v) \in \mathit{result}\ \mathcal{G}'$$

That is, starting from the entry point, the empty environment, and the empty trace, the result set of $\mathcal{G}$ is the same as the result set of $\mathcal{G}'$. It is worth noting that this is a *partial correctness* property, which ignores the possibility of non-termination of the optimization.

3 The TRANS Language

Now we have enough background to define the TRANS language itself. We will present here an overview of the syntax and semantics of the language, focusing on the differences between our formulation and its original presentation; for further details, see Kalvala et al. [6].

3.1 Overview

The basic units of TRANS are conditional graph rewrites of the form $A_1, A_2, ..., A_n$ if ϕ, where the A_i's are *actions* to be performed on a graph, and ϕ is a CTL-based *side condition*. Both the action and the condition may contain *metavariables*, which are instantiated with program objects when the transformation is applied. Actions are defined by

$$A ::= \mathsf{add\_edge}(n, m, t) \mid \mathsf{remove\_edge}(n, m, t) \mid \mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m \mid \mathsf{split\_edge}(n, m, t, p)$$

The action $\mathsf{add\_edge}(n, m, t)$ adds an edge labeled t between n and m; $\mathsf{remove\_edge}(n, m, t)$ removes an edge with label t between n and m; $\mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m$, replaces the instruction at n with the instructions $p_1, ..., p_m$; and $\mathsf{split\_edge}(n, m, t, p)$, inserts a node with instruction p in the middle of the edge from n to m labeled t. Many common program transformations can be expressed as a combination of these basic actions; in fact, many, including conversion to SSA form, can be expressed using only the replace action. Rather than using actual (graph) instructions, these actions take as arguments *instruction patterns*, in which metavariables can be used in place of program objects.

The side condition of a rewrite is an expression in CTL over paths in the graph, with the atomic predicates $node(n)$, which asserts that the metavariable n matches the current node, and $stmt(p)$, which asserts that the instruction pattern p matches the instruction label at the current node. These atomic predicates can be combined with the usual propositional connectives, as well as several CTL-specific operators [4], which fall into one of two categories. The first contains the *next-time* operators EX and AX, which assert that a statement holds on some successor and all successors of the current node respectively. The second group includes the *until* operators $E\phi_1 U \phi_2$ and $A\phi_1 U \phi_2$, which assert that ϕ_1 holds on all nodes until a node is reached on which ϕ_2 holds, along some path and all paths forward from the current node respectively. TRANS also uses reversed versions of the temporal operators, e.g. $\overleftarrow{EX}$, which make analogous assertions on nodes/paths backward from the current node. The formula $\phi@n$ asserts that ϕ holds starting from the node n. In addition to the CTL formulae on paths, TRANS side conditions include various basic predicates, such as $freevar(x, e)$, which asserts that x is a free variable of the expression e, or $fresh(x)$, which asserts that x does not appear in any instruction in the graph under consideration.

In order to determine the concrete transformation represented by a rewrite, a *valuation*, or map from metavariables to program objects, must be provided. This valuation, usually denoted by σ, is applied to both the action, to determine the actual nodes and instructions to be rewritten, and the side condition, to ensure that it produces a valid instance of the rewrite. A concrete instruction is obtained from a pattern p by applying the function $subst(\sigma, p)$, which replaces all the metavariables in p with the values given to them by σ.

A TRANS expression, then, is a group of conditional rewrites combined with any number of *strategies*, defined by

$$T ::= A_1, ..., A_m \text{if} \phi \mid \text{MATCH } \phi \text{ IN } T \mid T_1 \text{ THEN } T_2 \mid T_1 \square T_2 \mid \text{APPLY\_ALL } T$$

The TRANS expression MATCH ϕ IN T executes T under the restriction that its valuation must satisfy the condition ϕ; T_1 THEN T_2 applies T_1 and T_2 in sequence to a graph; $T_1 \square T_2$ applies either T_1 or T_2; and APPLY_ALL T recursively applies T under any possible valuation until it is no longer applicable.

Two aspects of the TRANS language are provided as parameters. The first parameter is the underlying language used to label the nodes of the CFGs; for instance, TRANS on L_0 CFGs has instruction patterns such as $x := e$ and `if` e. The second parameter is the set of basic predicates and atomic propositions used in the side conditions. Any property of a valuation and/or a CFG can serve as a predicate; any property of a valuation, a CFG, and a node can be used as an atomic proposition. We will modify both of these parameters when applying TRANS to the SSA transformation. Note that even TRANS on L_0 can express a variety of common optimizations; see Kalvala [6] for several examples, including dead code elimination and strength reduction.

3.2 Transformation Semantics

For the most part, the semantics of our definition of TRANS are identical to those given by Kalvala [6], with the addition of a few new basic predicates and local (defined) predicates. However, we take a simpler approach to the semantics of top-level transformations, and offer new definitions for the sequencing strategies THEN and APPLY_ALL which we believe are more consistent with the intended use of the strategies.

In the original presentation of TRANS, the semantics of a transformation were given by a semantic function $\lceil . \rceil : Transformation \to (PartValuation \times FlowGraph \to \mathbb{P}(FlowGraph \to FlowGraph))$ taking a transformation, a partial valuation (a partial function from metavariables to objects), and a graph, and giving a set of functions on graphs. That is, given a partial valuation and a graph, a transformation defined a set of functions to be applied to the graph. However, these functions are all intended to be applied to the graph provided, and are not guaranteed to be safe on any other graph (since the conditions of the transformation are checked on the original graph). Thus, the semantic function for a transformation can equally return the set of graphs resulting from the application of the transformation to the graph provided. We will take this approach in the following definitions, and use a semantic function $\lceil . \rceil : Transformation \to (PartValuation \times FlowGraph \to \mathbb{P}(FlowGraph))$.

Our second modification to the semantics deals with the compositional strategies THEN and APPLY_ALL. Originally, the semantics of THEN were given by

$$\lceil T_1 \text{ THEN } T_2 \rceil(\tau, \mathcal{G}) = \{f \circ g \mid f \in \lceil T_1 \rceil(\tau, \mathcal{G}) \wedge g \in \lceil T_2 \rceil(\tau, \mathcal{G})\}$$

Intuitively, both T_1 and T_2 are evaluated on the graph $\mathcal{G}$, and then the resulting functions are composed. This has the disadvantage that it violates one of the desired properties of TRANS, namely, that a transformation is only applied when its side condition is satisfied. Since T_2 is evaluated on $\mathcal{G}$, and then applied to $f(\mathcal{G})$ for some $f \in \lceil T_1 \rceil(\tau, \mathcal{G})$, the transformation it defines may be applied to

a graph on which its side condition does not hold. Using our modified semantic function, we propose the alternate definition

$$\lceil T_1 \text{ THEN } T_2 \rceil(\tau, \mathcal{G}) = \{\mathcal{G}'' \mid \exists \mathcal{G}'.\mathcal{G}' \in \lceil T_1 \rceil(\tau, \mathcal{G}) \wedge \mathcal{G}'' \in \lceil T_2 \rceil(\tau, \mathcal{G}')\}$$

Under this definition, T_2 is evaluated not on the original graph $\mathcal{G}$, but rather on the graph $\mathcal{G}'$ to which it will be applied, restoring the link between the condition and the transformation.

Our treatment of the APPLY_ALL strategy is similar, but more complex. The intention of the strategy is to apply a transformation everywhere in the graph, i.e., until the transformation can no longer be applied. The original semantics given for APPLY_ALL were

$$\lceil \text{APPLY\_ALL } T \rceil(\tau, \mathcal{G}) = \{f_1 \circ f_2 \circ ... \circ f_n \mid f_i \in \lceil T \rceil(\tau, \mathcal{G}) \setminus \{f_1, ..., f_{i-1}\}\}$$

This definition again allows the application of a transformation to a graph on which the side condition has not been checked, and also suffers from the problem that $\lceil T \rceil(\tau, \mathcal{G})$ may not be finite (or even countable). As a matter of fact, since the graph under consideration changes after each application of T, and the condition must be re-evaluated, the APPLY_ALL construct is essentially recursive. Thus, in order to give it the desired semantics, we must give it an inductive definition. We begin by defining an inductive set *apply_some* containing all graphs produced from applying T some number of times (possibly zero) to $\mathcal{G}$:

$$\mathcal{G} \in \mathit{apply\_some}(T, \tau, \mathcal{G})$$

$$\frac{\mathcal{G}' \in \lceil T \rceil(\tau, \mathcal{G}) \qquad \mathcal{G}'' \in \mathit{apply\_some}(T, \tau, \mathcal{G}')}{\mathcal{G}'' \in \mathit{apply\_some}(T, \tau, \mathcal{G})}$$

Then, by removing all of the intermediate graphs, we can define APPLY_ALL as yielding exactly the set of graphs in which T has been performed until it can no longer be applied:

$$\lceil \text{APPLY\_ALL } T \rceil(\tau, \mathcal{G}) = \mathit{apply\_some}(T, \tau, \mathcal{G}) \setminus \{\mathcal{G}' \mid \exists \mathcal{G}''.\mathcal{G}' \neq \mathcal{G}'' \wedge \mathcal{G}'' \in \lceil T \rceil(\tau, \mathcal{G}')\}$$

Under this definition, we once again have the property that a transformation is never applied to a graph on which its condition has not been evaluated, and we also know that T will no longer be applicable to the resulting graphs[4]. As part of the implementation of the framework, we have formalized the syntax and modified semantics of TRANS in Isabelle; the algebraic definitions can be translated directly into Isabelle datatypes and functions.

4 Theoretical Properties of TRANS

Although the TRANS approach has not previously been used to verify optimizations, the modularity provided by strategies and the use of CTL side conditions

[4] Note that if the recursive application of T is non-terminating, the set defined by APPLY_ALL T is empty.

make it well suited for formal verification. In this section, we state and sketch the proof of several simple but powerful properties of TRANS expressions, dealing with the effects of strategies and some common transformations. These lemmas also suggest a general methodology for verifying optimizations using TRANS: the first lemma can be used to break an optimization down into component transformations, and the following lemmas provide useful facts about common basic transformations.

When verifying a transformation, we frequently want to prove that some property of the program under consideration is *preserved*; that is, if it holds for the original program, then it also holds for the transformed program. Formally, we say that a transformation T preserves a property P of graphs if for all partial valuations τ and graphs $\mathcal{G}$, if P holds on $\mathcal{G}$, then for all result graphs $\mathcal{G}' \in \lceil T \rceil(\tau, \mathcal{G})$, P holds on $\mathcal{G}'$. We expect that if a group of transformations preserves some property, then any combination of those transformations also preserves the property. In fact, we can prove that the various strategies preserve any property that their component transformations preserve:

Lemma 1. *Suppose that T and T' preserve some property P. Then* MATCH ϕ IN T, T THEN T', $T \square T'$, *and* APPLY_ALL T *preserve P.*

Proof. For every strategy other than APPLY_ALL, the result follows directly from the semantics of the strategy. We can show that APPLY_ALL T preserves P by induction on the number of applications of T. □

This result allows us to break down the problem of showing that a complex transformation preserves a property into one of showing that each of its individual components (sub-expressions of the form $A_1, ..., A_m$ if ϕ) preserves the property. In the case study below we will use this lemma to show that a transformation preserves recoverability, but it could be used for any invariant on a graph.

The simplest transformation is one of the form replace n with p, where p is a single instruction pattern. This is the TRANS method of modifying a single instruction in the graph. Transformations of this form have many useful properties: for instance, they do not change the nodes, edges, or successor relation of a graph. In addition, we can give a simple condition under which such transformations preserve recoverability.

Lemma 2. *Consider a recoverable graph $\mathcal{G} = (N, E, I, S)$, and a valuation σ. If $I_{\mathcal{G}}\ \sigma(n)$ is the same type of instruction*[5] *as p, then $\mathcal{G}' = \lceil$replace n with $p\rceil(\sigma, \mathcal{G})$ is recoverable.*

Proof. The only effect of the single-instruction replace is to replace the instruction at $\sigma(n)$ with $subst(\sigma, p)$. Since the two instructions have the same type, this replacement does not affect the recoverability of the graph. □

This is a powerful lemma because it can be proved at the level of actions, and thus applies to any transformation using an action of this form. In other words, any

[5] I.e., they are both `if`-statements, or both `goto`-statements, etc.

transformation which performs only replacements of instructions by instructions of the same type is guaranteed to preserve recoverability. As we will see in our case study, some surprisingly complex transformations fall into this category.

A slightly more complicated transformation is one of the form replace n with $p_1, ..., p_m$, which replaces an instruction with a list of instructions. In this case, the nodes, edges, and successor function are all affected, but in a strictly local way. Once again, we can identify a common case in which such transformations preserve recoverability: that of inserting a list of assignments before some point in the program.

Lemma 3. *Consider a recoverable graph $\mathcal{G} = (N, E, I, S)$, and a valuation σ. If $\sigma(i) = I_{\mathcal{G}}\ \sigma(n)$ and $p_1, ..., p_m$ are all assignment patterns, then $\mathcal{G}' = \lceil$replace n with $p_1, ..., p_m, i\rceil(\sigma, \mathcal{G})$ is recoverable.*

Proof. The effect of this replace action is to insert a sequence of nodes before $\sigma(n)$. Since each inserted node is connected to its successor with a *seq*-edge, which is consistent with any assignment instruction, $\mathcal{G}'$ is still a CFG, and since no `ret` instructions are added or removed, $\mathcal{G}'$ is still recoverable. □

Note that the p_i's must be assignments because the edges inserted between the new nodes are labeled with *seq*. In fact, this lemma can be generalized to any instruction type for which the appropriate outgoing-edge set is a single *seq*-edge.

Lemma 4. *Consider a recoverable graph $\mathcal{G} = (N, E, I, S)$, and a valuation σ. If $\sigma(i) = I\ \mathcal{G}\ \sigma(n)$ and $p_1, ..., p_m$ are patterns consistent with an outgoing-edge set of a single seq-edge, then $\mathcal{G}' = \lceil$replace n with $p_1, ..., p_m, i\rceil(\sigma, \mathcal{G})$ is recoverable.*

In the case of L_0, these two statements are equivalent, but in more comprehensive languages, such as the L_1 language presented below, the more general statement of the lemma will be more useful.

5 The SSA Transformation

5.1 Static Single Assignment Form

One common program transformation in optimizing compilers is conversion to SSA form. While not in itself an optimization, this conversion allows for the application of various other optimizations and analyses [3,1]. A program in SSA form, as its name suggests, has no more than one assignment statement for each variable in the program. This is accomplished by labeling each instance of each variable with a subscript, or *index*; a unique index is given to each definition of a variable, and each use is given the index of the definition that reaches it. At join-points in the program, where two or more branches converge, there may be more than one reaching definition of a given variable; in this case, a *ϕ-function* is inserted at that point. The ϕ-function takes as arguments all the indexed instances of the variable that could reach the current point, chooses the

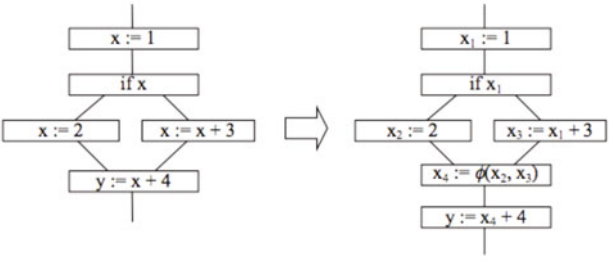


Fig. 1. Converting a CFG to SSA form

appropriate instance based on the trace of the current execution, and assigns it to a new instance of the variable. In this way, for every use of a variable in any instruction other than a ϕ-function, there is exactly one reaching definition, and so every variable can be labeled unambiguously with an index.

From a verification perspective, the SSA transformation is particularly interesting because 1) it introduces new variables into the program, and 2) it extends the language of the program by adding ϕ-functions. As such, we decided to test the power of the framework by using it to state and prove the correctness of SSA.

5.2 Defining SSA in TRANS

TRANS on L_0 is not sufficient to express the SSA transformation. However, we can extend L_0 to a new language L_1 that can support SSA by adding two new constructs: an indexed variable var_{num}, which can be used anywhere a variable can be used, and a ϕ-instruction $var := \phi(var, ..., var)$. The appropriate outgoing edge set for a ϕ-instruction is a single node with label *seq*, as in the case of an ordinary L_0 assignment instruction. The small-step evaluation rules for the ϕ-instruction are as follows, where *find_in_trace* $v_1, ..., v_n\ t\ \mathcal{G}$ gives the most recent definition of any of the x_i's in the trace t:

$$\frac{I_{\mathcal{G}}(l) = x := \phi(v_1, ..., v_n) \qquad \mathit{find\_in\_trace}\ v_1, ..., v_n\ t\ \mathcal{G} = v' \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{seq})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m[x \leftarrow m(v')], l', l; t)}$$

$$\frac{I_{\mathcal{G}}(l) = x := \phi(v_1, ..., v_n) \qquad \forall v'. \mathit{find\_in\_trace}\ v_1, ..., v_n\ t\ \mathcal{G} \neq v' \qquad \mathit{out\_edges}\ \mathcal{G}\ l = \{(l, l', \mathit{seq})\}}{(m, l, t) \rightarrow_{\mathcal{G}} (m, l', l; t)}$$

Note that we now make use of the execution trace included in the configuration. We include a rule to handle the corner case in which none of the variables in the ϕ-function are defined; in this case, the ϕ-function has no effect on the memory. We chose this outcome, rather than allowing the execution to become stuck, to most closely mimic the behavior of the original graph: if a variable was undefined, the program would not become stuck until it reached a use of the variable, and inserting a ϕ-function should not force a crash in a program that would not originally have crashed.

We also add several SSA-specific basic predicates to TRANS, such as *fresh_new*(x, j). The predicate *fresh_new*(x, j) holds when x is a variable, j is a number, and the indexed variable x_j does not appear anywhere in the program. We can use CTL on CFGs to define useful propositions on nodes such as *reaches*(x, j), which holds when x_j reaches the current node, and *multi_defs*(x), which holds when two different instances of x reach the current node.

We can now express the SSA conversion in TRANS on L_1. We break the conversion down into four stages, as follows: *add_index* adds indices to the left-hand side of each assignment in the program. *add_phi* inserts ϕ-functions (with empty bodies) at every point reached by multiple definitions[6]. *update* updates each use of a variable with the index of the reaching instance, and fills in the bodies of ϕ-functions. Finally, *refactor* transforms each indexed variable (which is semantically equivalent to the variable obtained by dropping the index) to an entirely new variable, giving it a separate location in memory. The conversion can be written in TRANS as follows:

$$\begin{aligned} \mathit{add\_index} ::= {} & \mathsf{replace}\ n\ \mathsf{with}\ (x_k := e) \\ & \mathsf{if}\ (\mathit{varlit}(x)\ \wedge\ \mathit{stmt}(x := e)\ @\ n\ \wedge\ \mathit{fresh\_new}(x, k)) \end{aligned}$$

The first step of the conversion is the simplest: each definition of a variable x is assigned a unique index k, without making any other changes to the graph.

$$\begin{aligned} & \mathit{add\_phi} ::= \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (x_k := \phi(), i) \\ & \ \mathsf{if}\ (\mathit{stmt}(i)\ @\ n\ \wedge\ \mathit{multi\_defs}\ x\ j_1\ j_2\ @\ n\ \wedge\ \mathit{fresh\_new}\ x\ k\ \wedge \\ & \quad \exists n_1\ (\exists n_2\ ((\neg n_1\ \mathit{is}\ n_2)\ \wedge\ (\overleftarrow{EX}\ \mathit{node}(n_1)\ \wedge\ \overleftarrow{EX}\ \mathit{node}(n_2))\ @\ n)) \wedge \\ & \quad A(\exists y, s\ (\mathit{stmt}(y := \phi(s))\ \wedge\ \neg x\ \mathit{is}\ y))U(\neg\exists y, s\ (\mathit{stmt}(y := \phi(s))))\ @\ n) \end{aligned}$$

In the second step, we insert the ϕ-functions at the proper locations in the program. Whereas in traditional algorithmic descriptions of the transformation, an analysis must be performed to determine where to place ϕ-functions, the TRANS approach allows us to explicitly state the condition under which a function should be inserted. The second half of the condition ensures that the ϕ-functions are only inserted at join points, and that no more than one ϕ-function per variable is inserted at each join point.

$$\begin{aligned} \mathit{update} ::= {} & \mathrm{MATCH}\ (\mathit{reaches}\ x\ k\ @\ n)\ \mathrm{IN} \\ & (\mathsf{replace}\ n\ \mathsf{with}\ (y := e[x_k])\ \mathsf{if}\ \mathit{stmt}(y := e[x])\ @\ n\ \square \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (\mathtt{if}\ e[x_k])\ \mathsf{if}\ \mathit{stmt}(\mathtt{if}\ e[x])\ @\ n\ \square \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (\mathtt{ret}(e[x_k]))\ \mathsf{if}\ \mathit{stmt}(\mathtt{ret}(e[x]))\ @\ n\ \square \\ & \ \mathsf{replace}\ n\ \mathsf{with}\ (x_{k'} := \phi(x_k, s))\ \mathsf{if}\ (\mathit{stmt}(x_{k'} := \phi(s))\ @\ n \wedge x_k \notin s)) \end{aligned}$$

In the third step, each use of a variable is annotated with the index of the reaching definition. Each ϕ-function is also populated with the reaching instances of the variable to which it is assigned.

[6] Note that this formulation of the conversion may not compute a minimal number of ϕ-functions; this could be achieved by adding an additional condition to *add_phi*.

$$
\begin{array}{l}
\textit{refactor} ::= \\
\quad \text{MATCH } (\textit{fresh}(z) \wedge (\textit{stmt}(x_k := e) \vee \textit{stmt}(x_k := \phi(s))) \ @ \ n) \text{ IN} \\
\quad ((\textsf{replace } n \textsf{ with } (z := e) \textsf{ if } \textit{stmt}(x_k := e) \ @ \ n \ \square \\
\quad\ \textsf{replace } n \textsf{ with } (z := \phi(s)) \textsf{ if } \textit{stmt}(x_k := \phi(s)) \ @ \ n) \text{ THEN} \\
\quad\ \text{APPLY\_ALL } (\textsf{replace } m \textsf{ with} (y := f[z]) \textsf{ if } \textit{stmt}(y := f[x_k]) \ @ \ m \ \square \\
\qquad\qquad\qquad \textsf{replace } m \textsf{ with } (\texttt{if } f[z]) \textsf{ if} \textit{stmt}(\texttt{if } f[x_k]) \ @ \ m \ \square \\
\qquad\qquad\qquad \textsf{replace } m \textsf{ with } (\texttt{ret}(f[z])) \textsf{ if } \textit{stmt}(\texttt{ret}(f[x_k])) \ @ \ m \ \square \\
\qquad\qquad\qquad \textsf{replace } m \textsf{ with } (y := \phi(z, t)) \textsf{ if } \textit{stmt}(y := \phi(x_k, t)) \ @ \ m))
\end{array}
$$

The final step is to replace each indexed variable with an entirely new variable. While this is not explicitly a part of most SSA algorithms, it is necessary because of the memory model used in our implementation, in which every instance of a variable points to the same memory location. The advantage of this approach is that each individual step can be shown to preserve the program's semantics, allowing for a more modular proof of correctness, as will be shown below.

Each step performs the desired transformation for one variable, so we use the APPLY_ALL strategy to extend it to the entire program, giving the complete SSA transformation:

$$
\begin{array}{l}
\textit{conversion} ::= (\text{APPLY\_ALL } \textit{add\_index}) \text{ THEN } (\text{APPLY\_ALL } \textit{add\_phi}) \text{ THEN} \\
\qquad\qquad (\text{APPLY\_ALL } \textit{update}) \text{ THEN } (\text{APPLY\_ALL } \textit{refactor})
\end{array}
$$

5.3 Proving SSA Correct

Given the definition of SSA in TRANS, and the lemmas shown above, we are now ready to construct a proof of correctness for SSA. In fact, using the Isabelle implementation of the framework, we have done exactly that, proving that any graph produced by the conversion is semantically equivalent to the input graph. We will outline the proof below. Since we have taken the trouble to express the conversion in as modular a fashion as possible, we can verify it by proving appropriate theorems for each of the individual steps, and then combining them with the simple facts about the strategies shown above.

Thanks to the power of the framework, we can easily show that each step of the transformation preserves recoverability.

Lemma 5. *add_index, add_phi, update, and refactor all preserve recoverability.*

Proof. All of the actions in *add_index*, *update*, and *refactor* are of the form specified in Lemma 2, so they preserve recoverability. The action in *add_phi* is of the form specified in Lemma 4, so it also preserves recoverability. Thus, we can conclude that the conversion preserves recoverability. □

Corollary 1. *The complete SSA conversion preserves recoverability.*

Proof. Since recoverability is a property of a graph, by Lemma 1 the conversion preserves recoverability if each of its steps preserves recoverability. □

Now we can demonstrate the correctness of the transformation. Once again, we will proceed modularly, stating an appropriate lemma for each step of the conversion. Of course, the steps are not independent of each other; they are semantics-preserving only in combination. However, we can separate them by stating appropriate conditions that must hold for each step to be correct, and then showing that these conditions hold after the previous step. Ultimately, we will have shown not only that the results of the transformation are semantically equivalent to its input, but also that the resulting graphs are in SSA form. We can identify three properties that should hold at various points in the transformation:

1. The graph has no more than one definition for each variable instance it contains.
2. There is no more than one instance of each variable in the graph that reaches each node labeled with a non-ϕ instruction, and each instruction uses a variable instance x_j only if x_j reaches the node labeled with the instruction.
3. If an instance of a variable x_j reaches a node in the graph labeled with a ϕ-function, and another instance x_k is in the body of the ϕ-function, then x_j is also in the body of the ϕ-function.

Note that if a graph has no indexed variables, then property 1 is sufficient to imply that the graph is in SSA form, since there is at most one instance of each variable.

Lemma 6. *Consider a CFG $\mathcal{G}$ with no ϕ-functions. Then every graph in $\lceil add\_index \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$ and has no ϕ-functions. Furthermore, every graph in $\lceil$*APPLY_ALL *$add\_index \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has no ϕ-functions, and has property 1.*

Lemma 7. *Consider a recoverable CFG $\mathcal{G}$ with no non-empty ϕ-functions such that property 1 holds for $\mathcal{G}$. Then any graph in $\lceil add\_phi \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has no non-empty ϕ-functions, and has property 1. Any graph in $\lceil$*APPLY_ALL *$add\_phi \rceil(empty, \mathcal{G})$ also is semantically equivalent to $\mathcal{G}$, has no non-empty ϕ-functions, and has property 1.*

Lemma 8. *Consider a CFG $\mathcal{G}$ with property 1 such that every ϕ-function in $\mathcal{G}$ is of the form $x_j = \phi(x_{k1}, ..., x_{kn})$ – i.e., every variable in the body of the ϕ-function has the same base as the variable on the left-hand side. Then every graph in $\lceil update \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has property 1, and has only ϕ-functions of the form $x_j = \phi(x_{k1}, ..., x_{kn})$. Furthermore, every graph in $\lceil$*APPLY_ALL *$update \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has only ϕ-functions of the form $x_j = \phi(x_{k1}, ..., x_{kn})$, and has properties 1, 2, and 3.*

Lemma 9. *Consider a CFG $\mathcal{G}$ with properties 1, 2, and 3. Then every graph in $\lceil refactor \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$ and has properties 1, 2, and 3. Furthermore, every graph in $\lceil$*APPLY_ALL *$refactor \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has properties 1, 2, and 3, and has no indexed variables.*

Theorem 1. *Consider a recoverable L_0 CFG $\mathcal{G}$. Every graph in $\lceil conversion \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, and is in SSA form.*

Proof. We will proceed by breaking the conversion into steps, and then make use of the above lemmas. Since L_0 does not contain ϕ-functions, we know that $\mathcal{G}$ has no ϕ-functions. Thus, by Lemmas 5 and 6, every graph in $\lceil$APPLY_ALL *add_index*$\rceil(empty, \mathcal{G})$ is recoverable, is semantically equivalent to $\mathcal{G}$, has no ϕ-functions, and has property 1. If a graph has no ϕ-functions, then it certainly has no non-empty ϕ-functions. Thus, by Lemma 7, every graph in $\lceil$(APPLY_ALL *add_index* THEN (APPLY_ALL *add_phi*)$\rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has no non-empty ϕ-functions, and has property 1. If every ϕ-function in a graph is empty, then certainly every ϕ-function is of the form $x_j = \phi(x_{k1}, ..., x_{kn})$. Thus, by Lemma 8, every graph in

$$\lceil(\text{APPLY\_ALL } add\_index) \text{ THEN}$$
$$(\text{APPLY\_ALL } add\_phi) \text{ THEN } (\text{APPLY\_ALL } update)\rceil(empty, \mathcal{G})$$

is semantically equivalent to $\mathcal{G}$, and has properties 1, 2, and 3. Finally, by Lemma 9, we can conclude that every graph in $\lceil conversion \rceil(empty, \mathcal{G})$ is semantically equivalent to $\mathcal{G}$, has properties 1, 2, and 3, and has no indexed variables, implying that it is in SSA form. □

6 Conclusions and Further Research

We have outlined a new approach to proving the correctness of compiler optimizations, as well as an Isabelle-implemented framework to support this approach. We have clarified and formalized the semantics of the TRANS language, making it into a tool for use in constructing fully formal proofs of correctness for program transformations; the Isabelle code can be found online [10]. By using TRANS to state a transformation in terms of conditional rewrites on CFGs, a verifier can take advantage of the general lemmas we have established about the semantics of TRANS. We have demonstrated the generality of this approach by parameterizing TRANS to express the SSA transformation, and then presented a new and relatively concise proof of correctness for the transformation within our framework. This is, to the best of our knowledge, the first machine-assisted verification of an optimization expressed in TRANS, and the first verified SSA conversion of any sort. We hope that the approach demonstrated here will significantly reduce the difficulty of the problem of verifying optimizations.

The L_0 language on which our implementation of TRANS is based is relatively simple, and does not include some common features of intermediate languages, such as arrays and function calls. Our experience with L_1 suggests that parameterizing TRANS with more expressive languages can be easily accomplished; the more difficult aspect of adding new features is providing a semantics for programs with these features and proving related lemmas about TRANS. This would bring the framework a step closer to dealing with real-world intermediate languages, and enable it to handle a wider range of optimizations.

The tendency to leave compiler optimizations unverified might be defended with the argument that most optimizations have been in use for decades without major problems. However, bugs have been discovered even in C compilers [11], and the situation is even more complicated when dealing with concurrency. Most optimizations for parallel programs are experimental and have not been widely field-tested, and it is much more difficult to put forward a convincing informal justification for a parallel optimization. For this reason, we believe that compilers for parallel languages particularly stand to benefit from formal proofs of correctness. While formalisms such as control-flow graphs and CTL are sufficient for expressing and verifying optimizations on sequential programs, they are not as obviously applicable to the case of parallel programs. However, we believe that concurrent analogues of these formalisms, such as parallel program graphs [14] and alternating-time temporal logic [2], will allow us to extend our approach to parallel optimizations.

7 Related Work

This research builds on the work of Lacey et al. [7] and Kalvala et al. [6], who have defined the core concepts of the TRANS language and used it to express and (on paper) verify several optimizations.

One of the first computer-assisted verifications of a compiler was due to Moore [12]. The source language of this compiler was very low-level, and the compiler did not perform any optimizations.

Leroy [9] has developed a Coq-based framework for the verification of optimizations that depend on dataflow analysis. Code transformations operate on instructions, and are verified by comparing the semantics of the transformed instructions to that of the original instructions under conditions provided by the dataflow analysis. However, this approach can handle only limited modifications to program structure. Visser et al. [16] use a rewrite system to define optimizations on programs in a functional language. Their system includes a variety of rewriting strategies, but does not deal with conditional rewriting or verification.

Translation validation [13,15] is another method of verifying compiler optimizations. In this approach, rather than proving the optimization correct for all programs, an automatic verifier is used on the results of each application of the optimization to ensure that the resulting program has the same semantics as the original program. Since the process is fully automated, the verification process is considerably more lightweight, but the range of optimizations that can be handled is more limited.

Various work has been done on the formal properties of code already in SSA form; see for instance Blech and Glesner [5], who have used Isabelle to verify an algorithm for code generation from SSA.

Acknowledgments. We would like to thank Sara Kalvala and Richard Warburton for introducing us to TRANS and clarifying various points.

References

1. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 1–11. ACM, New York (1988)
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
3. Appel, A.W.: Modern Compiler Implementation in ML. Cambridge University Press, New York (2004)
4. Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. In: POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 164–176. ACM, New York (1981)
5. Blech, J.O., Glesner, S.: A formal correctness proof for code generation from ssa form in isabelle/hol. In: Proceedings der 3. Arbeitstagung Programmiersprachen (ATPS) auf der 34. Jahrestagung der Gesellschaft für Informatik, September 2004. Lecture Notes in Informatics (2004), http://www.info.uni-karlsruhe.de/papers/Blech-Glesner-ATPS-2004.pdf
6. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Trans. Program. Lang. Syst. 31(4), 1–48 (2009)
7. Lacey, D., Jones, N.D., Van Wyk, E., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 283–294. ACM, New York (2002)
8. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 42–54. ACM, New York (2006)
9. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (2009)
10. Mansky, W.: TRANS in Isabelle, http://www.cs.illinois.edu/homes/mansky1
11. McKeeman, W.M.: A formally verified compiler backend. Digital Technical Journal 10(1), 100–107 (1998)
12. Moore, J.S.: A mechanically verified language implementation. J. Autom. Reason. 5(4), 461–492 (1989)
13. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
14. Sarkar, V.: Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In: Huang, C.-H., Sadayappan, P., Sehr, D. (eds.) LCPC 1997. LNCS, vol. 1366, pp. 94–113. Springer, Heidelberg (1998)
15. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 17–27. ACM, New York (2008)
16. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. SIGPLAN Not. 34(1), 13–26 (1999)

On the Formalization of the Lebesgue Integration Theory in HOL

Tarek Mhamdi, Osman Hasan, and Sofiène Tahar

ECE Department, Concordia University, Montreal, QC, Canada
{mhamdi,o_hasan,tahar}@ece.concordia.ca

Abstract. Lebesgue integration is a fundamental concept in many mathematical theories, such as real analysis, probability and information theory. Reported higher-order-logic formalizations of the Lebesgue integral either do not include, or have a limited support for the Borel algebra, which is the canonical sigma algebra used on any metric space over which the Lebesgue integral is defined. In this paper, we overcome this limitation by presenting a formalization of the Borel sigma algebra that can be used on any metric space, such as the complex numbers or the n-dimensional Euclidean space. Building on top of this framework, we have been able to prove some key Lebesgue integral properties, like its linearity and monotone convergence. Furthermore, we present the formalization of the "almost everywhere" relation and prove that the Lebesgue integral does not distinguish between functions which differ on a null set as well as other important results based on this concept. As applications, we present the verification of Markov and Chebyshev inequalities and the Weak Law of Large Numbers theorem.

1 Introduction

Formal modeling of physical systems or devices is not a very straightforward task due to the presence of many continuous and unpredictable components. For example, embedded systems are operating in a concrete physical environment with continuous dynamics; cryptography heavily relies upon information theoretic concepts; a broad area of chemistry and biology (and biophysics) worries about stochastic effects and phenomena, etc. Formal models of computation have in the past mostly been considered independent of the continuous or unpredictable world. In classical formal verification efforts, hardware and software are viewed as discrete models of computation. But due to the dire need of accurate analysis in safety-critical domains, there is a growing trend towards incorporating continuous and unpredictable physical realities in the formal models of physical systems.

Lebesgue integration [1] is a fundamental concept in many mathematical theories, such as real analysis [5], probability [6] and information theory, which are widely used to model and reason about the continuous and unpredictable components of physical systems. The reasons for its extensive usage, compared to the commonly known Riemann integral, include the ability to handle a broader

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 387–402, 2010.

class of functions, which are defined over more general types than the real line, and its better behavior when it comes to interchanging limits and integrals. In order to facilitate the formal analysis of physical systems, two higher-order-logic formalizations of the Lebesgue integral have been recently reported [3,15]. However, they either do not include, or have a very limited support for the Borel algebra [2], which is a sigma algebra generated by the open sets. These deficiencies restrict the formal reasoning about some very useful Lebesgue integral properties, which in turn limits the scope of formally analyzing physical systems.

In this paper, we present a generalized formalization of the Lebesgue integral in order to exploit its full potential for the formal analysis of other systems. We first formalize the Borel algebra that provides a unified framework to prove the Lebesgue integral properties and measurability theorems on any metric space, such as the real numbers, the complex numbers or the n-dimensional Euclidean space. Building on top of this formalization, we prove some of the key Lebesgue integral properties as well as its convergence theorems. Similarly, we formalize the notion of "almost everywhere" [1] and prove that the Lebesgue integral does not distinguish between functions which differ on a null set along with some other useful results based on the "almost everywhere" relation. In order to illustrate the practical effectiveness of our work, we utilize it to verify the Chebyshev and Markov inequalities and the Weak Law of Large Numbers (WLLN) [14], which are widely used properties in probability and information theories.

We used the HOL theorem prover for the above mentioned formalization and verification tasks. The main motivation behind this choice was to build upon existing formalizations of measure [10] and Lebesgue integration [3] theories.

The rest of the paper is organized as follows: Section 2 provides a review of related work. In Section 3, we give an overview of main definitions of the measure theory [2]. Section 4 presents our formalization of the Borel theory, which is used in Section 5 to prove the main properties of the Lebesgue integral and its convergence theorems. In Section 6, we use our formalization for verifying some important theorems from the theory of probability. Finally, Section 7 concludes the paper and provides hints to future work.

2 Related Work

Coble [3] generalized the measure theory formalization by Hurd [10] and built on it to formalize the Lebesgue integration theory. He proved some properties of the Lebesgue integral but only for the class of positive simple functions. Besides, multiple theorems in Coble's work have the assumption that every set is measurable which is not correct in most cases of interest. We propose to prove the Lebesgue integral properties and convergence theorems for arbitrary functions by providing a formalization of the Borel sigma algebra, which has also been used to overcome the assumption of Cobles's work.

Based on the work of Hurd [10], Richter [15] also formalized the measure theory in Isabelle/HOL, where he restricts the measure spaces that can be constructed. In Richter's formalization, a measure space is the pair $(\mathcal{A}, \mu)$; $\mathcal{A}$ is a set

of subsets of X, called the set of measurable sets and μ is a measure function. The space is implicitly the universal set of the appropriate type. This approach does not allow to construct a measure space where the space is not the universal set. The only way to apply this approach for an arbitrary space X is to define a new type for the elements of X, redefine operations on this set and prove properties of these operations. This requires considerable effort that needs to be done for every space of interest. The work we propose in this paper is based on the formalization of Coble [3] where we define a measure space as a triplet $(X, \mathcal{A}, \mu)$; the set X being the space.

Richter [15] defined the Borel sets as being generated by the intervals. In the formalization we propose in this paper, the Borel sigma algebra is generated by the open sets and is more general as it can be applied not only to the real numbers but to any metric space such as the complex numbers or $\mathbb{R}^n$, the n-dimensional Euclidean space. It provides a unified framework to prove the measurability theorems in these spaces. Besides, our formalization allows us to prove that any continuous function is measurable which is an important result to prove the measurability of a large class of functions, in particular, trigonometric and exponential functions. To prove this result we also formalize in this paper key concepts of topology [13] in HOL.

In his work in topology in the PVS theorem prover, Lester [11] provided formalizations for measure and integration theories but did not prove the properties of the Lebesgue integral nor its convergence theorems such as the Lebesgue Monotone Convergence.

3 Measure Theory

The measure theory formalization in HOL was essentially done in [10] and [3]. We make use of this formalization in our development and hence will only mention the main definitions. A measure is a way to assign a number to a set, interpreted as its size, a generalization of the concepts of length, area, volume, etc. A measure is defined on a class of subsets called the measurable sets. One important condition for a measure function is countable additivity, meaning that the measure of a countable collection of disjoint sets is the sum of their measures. This leads to the requirement that the measurable sets should form a sigma algebra.

Definition 1. *Let $\mathcal{A}$ be a collection of subsets of a space X. $\mathcal{A}$ defines a sigma algebra on X* iff *$\mathcal{A}$ contains the empty set $\emptyset$, and is closed under countable unions and complementation within the space X.*

Definition 1 is formalized in HOL as

```
⊢ ∀X A. sigma_algebra (X,A) =
    subset_class X A ∧ {} ∈ A ∧ (∀s. s ∈ A ⇒ X\s ∈ A) ∧
    ∀c. countable c ∧ c ⊆ A ⇒ ⋃c ∈ A
```

where $X\backslash s$ denotes the complement of s within X, $\bigcup c$ the union of all elements of c and `subset_class` is defined as

```
⊢ ∀X A. subset_class X A = ∀s. s ∈ A ⇒ s ⊆ X
```

A set S is countable if its elements can be counted one at a time, or in other words, if every element of the set can be associated with a natural number, i.e., there exists a surjective function $f : \mathbb{N} \to S$.

```
⊢ ∀s. countable s = ∃f. ∀x. x ∈ s ⇒ ∃n. f n = x
```

The smallest sigma algebra on a space X is $\mathcal{A} = \{\emptyset, X\}$ and the largest is its powerset, $\mathcal{P}(X)$, the set of all subsets of X. The pair $(X, \mathcal{A})$ is called a σ-field or a measurable space, $\mathcal{A}$ is the set of measurable sets.

For any collection G of subsets of X we can construct the smallest sigma algebra on X containing G, we call it the sigma algebra on X generated by G, denoted by $\sigma(X, G)$. There is at least one sigma algebra on X containing G, namely the power set of X. $\sigma(X, G)$ is the intersection of all those sigma algebras.

```
⊢ ∀X G. sigma X G = (X,⋂{s | G ⊆ s ∧ sigma_algebra (X,s)})
```

Definition 2. *A triplet* $(X, \mathcal{A}, \mu)$ *is a measure space* iff $(X, \mathcal{A})$ *is a measurable space and* $\mu : \mathcal{A} \to \mathbb{R}$ *is a non-negative and countably additive measure function.*

```
⊢ ∀X A mu. measure_space (X,A,mu) =
    sigma_algebra (X,A) ∧ positive (X,A,mu) ∧
    countably_additive (X,A,mu)
```

A probability space $(\Omega, \mathcal{A}, p)$ is a measure space satisfying $p(\Omega) = 1$.
There is a special class of functions, called measurable functions, that are structure preserving, in the sense that the inverse image of each measurable set is also measurable. This is analogous to continuous functions in metric spaces where the inverse image of an open set is open.

Definition 3. *Let* $(X_1, \mathcal{A}_1)$ *and* $(X_2, \mathcal{A}_2)$ *be two measurable spaces. A function* $f : X_1 \to X_2$ *is called measurable with respect to* $(\mathcal{A}_1, \mathcal{A}_2)$ *(or* $(\mathcal{A}_1, \mathcal{A}_2)$ *measurable)* iff $f^{-1}(A) \in \mathcal{A}_1$ *for all* $A \in \mathcal{A}_2$.

$f^{-1}(A)$ denotes the inverse image of A. The HOL formalization is the following.

```
⊢ ∀a b f.
    f ∈ measurable a b =
    sigma_algebra a ∧ sigma_algebra b ∧ f ∈ (space a → space b) ∧
    ∀s. s ∈ subsets b ⇒ PREIMAGE f s ∩ space a ∈ subsets a
```

In this definition, we did not specify any structure on the measurable spaces. If we consider a function f that takes its values on a metric space, most commonly the set of real numbers or complex numbers, then the Borel sigma algebra on that space is used.

4 Borel Theory

Working with the Borel sigma algebra makes the set of measurable functions a vector space. It also allows us to prove various properties of the measurable functions necessary for the development in HOL of the Lebesgue integral and its properties.

Definition 4. *The Borel sigma algebra on a space X is the smallest sigma algebra generated by the open sets of X.*

```
⊢ borel X = sigma X (open_sets X)
```

An important example, especially in the theory of probability, is the Borel sigma algebra on $\mathbb{R}$, denoted by $\mathcal{B}(\mathbb{R})$.

```
⊢ Borel = sigma UNIV (open_sets UNIV)
```

Clearly, to formalize as well as prove in HOL various properties of $\mathcal{B}(\mathbb{R})$, we need to formalize some topology concepts of $\mathbb{R}$ and also provide a formalization of the set of rational numbers $\mathbb{Q}$. A theory for the rational numbers was developed in HOL but does not include the theorems that we need and is in fact unusable for our development because we need to work on rational numbers as a subset of real numbers and not of a different HOL type. We will prove later that $\mathcal{B}(\mathbb{R})$ is generated by the open intervals. This was actually used in many textbooks as a starting definition for the Borel sigma algebra on $\mathbb{R}$. While we will prove that the two definitions are equivalent in the case of the real line, our formalization is vastly more general and can be used for any metric space such as the complex numbers or $\mathbb{R}^n$, the n-dimensional Euclidian space.

4.1 Rational Numbers

A rational number is any number that can be expressed as the quotient of two integers, the denominator of which is positive. We use natural numbers and express $\mathbb{Q}$, the set of rational numbers, as the union of non-negative ($\mathbb{Q}^+$) and non-positive ($\mathbb{Q}^-$) rational numbers.

$$\vdash \mathbb{Q} = \{r \mid \exists\ n, m.\ r = \tfrac{n}{m} \ \wedge\ m > 0\} \cup \{r \mid \exists\ n, m.\ r = \tfrac{-n}{m} \ \wedge\ m > 0\}$$

We prove in HOL an extensive number of reassuring properties on the set $\mathbb{Q}$ as well as few other less straightforward ones, namely, $\mathbb{Q}$ is countable, infinite and dense in $\mathbb{R}$.

Theorem 1. $\mathbb{N} \subset \mathbb{Q}$ *and* $\forall x, y \in \mathbb{Q}, -x,\ x + y,\ x - y,\ x * y\ \in \mathbb{Q}$ *and* $\forall y \neq 0$, $\frac{1}{y}$ *and* $\frac{x}{y}\ \in \mathbb{Q}$

A proof of this theorem in HOL is at the same time straightforward and tedious but it is necessary to manipulate elements of the newly defined set of rational numbers and prove their membership to $\mathbb{Q}$ in the following theorems.

Theorem 2. *The set of rational numbers $\mathbb{Q}$ is countable.*

Proof. We prove that there exists a bijection f_1 from the set of natural numbers $\mathbb{N}$ to the cross product of $\mathbb{N}$ and $\mathbb{N}^*$ ($f_1 : \mathbb{N} \to \mathbb{N} \times \mathbb{N}^*$). Let $f_2 : \mathbb{N} \times \mathbb{N}^* \to \mathbb{Q}^+$ such that $f_2(a, b) = \frac{a}{b}$. and $f = f_2 \circ f_1$. Then $\forall x \in \mathbb{Q}^+$, there exists $n \in \mathbb{N}$ such that $f(n) = x$. This proves that $\mathbb{Q}^+$ is countable. Similarly, we prove that $\mathbb{Q}^-$ is countable and that the union of two countable sets is countable. □

Theorem 3. *($\mathbb{Q}$ dense in $\mathbb{R}$)*
$\forall x, y \in \mathbb{R}$ and $x < y$, there exists $r \in \mathbb{Q}$ such that $x < r < y$.

Proof. We start by defining the ceiling of x as the smallest natural number larger than x, denoted by $\lceil x \rceil$ and prove that $\forall x,\ x \leq \lceil x \rceil$ and $\forall x \geq 0, \lceil x \rceil < x + 1$. Let $x, y \in \mathbb{R}$ such that $x < y$. We use the ceiling function and the Archimedean property to construct r such that $x < r < y$. □

Another definition that will be useful in our development is the set of open intervals with rational end-points $I_r = \{]r_1, r_2[: r_1, r_2 \in \mathbb{Q}\}$.

```
⊢ open_intervals_set = {{x | a<x ∧ x<b} | a ∈ UNIV ∧ b ∈ UNIV}
```

We prove that I_r is countable by showing that the mapping $I_r \to \mathbb{Q}\times\mathbb{Q}$ that sends an open interval $]r_1, r_2[\in I_r$ to the ordered pair of rational numbers $(r_1, r_2) \in \mathbb{Q} \times \mathbb{Q}$ is injective, and that the cross product of two countable sets, $\mathbb{Q}$ in this case, is countable.

4.2 Topology

To define the Borel sigma algebra on $\mathbb{R}$, we need some concepts of the topology of $\mathbb{R}$ formalized in HOL. Some of this was already developed by Harrison [7] but his formalization in HOL does not use the set theory and also lacks some of the important theorems that we need in our development. Harrison, later, developed an extensive topology theory [8] in HOL-Light. In the following, we define the concepts of neighborhood and open set in $\mathbb{R}$ and prove the required theorems.

Definition 5. *Let $a \in A \subset \mathbb{R}$. A is a neighborhood of a* iff *there exists a real number $d > 0$ such that $\forall x.\ |x - a| < d \Rightarrow\ x \in A$. In other words, a is an interior point of A.*

```
⊢ ∀A a.
    neighborhood_R A a = ∃d. 0<d ∧ ∀y. a - d<y ∧ y<a + d ⇒ y ∈ A
```

Definition 6. *A set that is a neighborhood to all of its points in an open set. Equivalently, if every point of a set is an interior point then the set is open.*

```
⊢ ∀A. open_set_R A = ∀x. x ∈ A ⇒ neighborhood_R A x
```

Theorem 4. *The empty set and the universal set are open.*

Theorem 5. *Every open interval is an open set.*

Theorem 6. *The union of any family of open sets is open. The intersection of a finite number of open sets is open.*

Theorem 7. *Every open set in* $\mathbb{R}$ *is the union of a countable family of open intervals.*

Proof. We only show the proof for Theorem 7. Let A be an open set in $\mathbb{R}$, then by the definition of open set, for all x in A there exists an open interval containing x such that $]a, b[\subset A$. Using the property of density of $\mathbb{Q}$ in $\mathbb{R}$, there exists $]a_r, b_r[\subset A$ containing x, a_r and b_r being rational numbers. A is the union of family of elements of I_r which is then countable because I_r is countable. □

Theorem 8. *The inverse image of an open set by a continuous function is open.*

Proof. Let A be an open set in $\mathbb{R}$. From the previous theorem, A is a countable union of open intervals (A_i). $f^{-1}(A) = f^{-1}(\bigcup A_i) = \bigcup f^{-1}(A_i)$. Using Theorem 6, it suffices to prove that the inverse image of an open interval is open. For this we use the definition of a continuous function and the limit of a function to prove that any point of $f^{-1}(A_i)$ is an interior point. □

4.3 Borel Measurable Sets

In this section, we prove in HOL that the Borel algebra on the real line $\mathcal{B}(\mathbb{R})$ is generated by the open intervals $(]c, d[$ for $c, d \in \mathbb{R})$. We show that it is also generated by any of the following classes of intervals: $]-\infty, c[$, $[c, +\infty[$, $]c, +\infty[$, $]-\infty, c]$, $[c, d[$, $]c, d]$, $[c, d]$, where $c, d \in \mathbb{R}$.

Theorem 9. $\mathcal{B}(\mathbb{R})$ *is generated by the open intervals* $]c, d[$ *where* $c, d \in \mathbb{R}$

Proof. The sigma algebra generated by the open intervals, σ_I, is by definition the intersection of all sigma algebras containing the open intervals. $\mathcal{B}(\mathbb{R})$ is one of them because the open intervals are open sets (Theorem 5). Hence, $\sigma_I \subseteq \mathcal{B}(\mathbb{R})$. Conversely, $\mathcal{B}(\mathbb{R})$ is the intersection of all the sigma algebras containing the open sets. σ_I is one of them because every open set on the real line is the union of a countable collection of open intervals (Theorem 7). Consequently $\mathcal{B}(\mathbb{R}) \subseteq \sigma_I$ and finally $\mathcal{B}(\mathbb{R}) = \sigma_I$.

To prove that $\mathcal{B}(\mathbb{R})$ is also generated by the other classes of intervals, it suffices to prove that any interval]a,b[is contained in the sigma algebra corresponding to each class. For the case of the intervals of type $[c, d[$, this follows from the equation $]a, b[= \bigcup_n [a + \frac{1}{2^n}, b[$.

For the open rays $]-\infty, c\,[$, the result follows from the fact that $[a, b[$ can be written as the difference of two rays, $[a, b[=]-\infty, b\,[\setminus]-\infty, a\,[$.
In a similar manner, we prove in HOL that all mentioned classes of intervals generate the Borel sigma algebra on $\mathbb{R}$. □

Another useful result, asserts that the singleton sets are measurable sets of $\mathcal{B}(\mathbb{R})$.

Theorem 10. $\forall c \in \mathbb{R},\ \{c\} \in \mathcal{B}(\mathbb{R})$

The proof of this theorem follows from the fact that a sigma algebra is closed under countable intersection and the equation

$$\forall c \in \mathbb{R} \quad \{c\} = \bigcap_n [c - \frac{1}{2^n}, c + \frac{1}{2^n}[$$

4.4 Real Valued Measurable Functions

Recall that in order to check if a function f is measurable with respect to $(\mathcal{A}_1, \mathcal{A}_2)$, it is necessary to check that for any $A \in \mathcal{A}_2$, its inverse image $f^{-1}(A) \in \mathcal{A}_1$. The following theorem states that, for real-valued functions, it suffices to perform the check on the open rays $(]-\infty, c[,\ c \in \mathbb{R})$.

Theorem 11. *Let $(X, \mathcal{A})$ be a measurable space. A function $f : X \to \mathbb{R}$ is measurable with respect to $(\mathcal{A}, \mathcal{B}(\mathbb{R}))$ iff $\forall c \in \mathbb{R},\ f^{-1}(]-\infty, c[) \in \mathcal{A}$*

Proof. Suppose that f is measurable with respect to $(\mathcal{A}, \mathcal{B}(\mathbb{R}))$, we showed in the previous section that $\forall c \in \mathbb{R},]-\infty, c[\in \mathcal{B}(\mathbb{R})$. Since f is measurable then $f^{-1}(]-\infty, c[) \in \mathcal{A}$. Now suppose that $\forall c \in \mathbb{R},\ f^{-1}(]-\infty, c[) \in \mathcal{A}$, we need to prove $\forall A \in \mathcal{B}(\mathbb{R}),\ f^{-1}(A) \in \mathcal{A}$. This follows from Theorem 7 stating that A is a countable union of open intervals and the equalities $f^{-1}(\bigcup_{n\in\mathbb{N}} A_n) = \bigcup_{n\in\mathbb{N}} f^{-1}(A_n)$ and $f^{-1}(]-\infty, c[) = \bigcup_{n\in\mathbb{N}} f^{-1}(]-n, c[)$ □

In a similar manner, we prove in HOL that f is measurable with respect to $(\mathcal{A}, \mathcal{B}(\mathbb{R}))$ iff $\forall c, d \in \mathbb{R}$ the inverse image of any of the following classes of intervals is an element of $\mathcal{A}$: $]-\infty, c[$, $[c, +\infty[$, $]c, +\infty[$, $]-\infty, c]$, $[c, d[$, $]c, d]$, $[c, d]$.

Every constant real function on a space X is measurable. In fact, if $\forall x \in X,\ f(x) = k$, then if $c \leq k,\ f^{-1}(]-\infty, c[) = \emptyset \in \mathcal{A}$. Otherwise $f^{-1}(]-\infty, c[) = X \in \mathcal{A}$. The indicator function on a set A is measurable iff A is measurable. In fact, $I_A^{-1}(]-\infty, c[) = \emptyset$, X or $X \backslash A$ when $c \leq 0$, $c > 1$ or $0 < c \leq 1$ respectively. We prove in HOL various properties of the real-valued measurable functions.

Theorem 12. *Let f and g be measurable functions and $c \in \mathbb{R}$ then the following functions are also measurable: $cf, |f|, f^n, f + g, fg$ and $max(f, g)$.*

Theorem 13. *If (f_n) is a sequence of real-valued measurable functions such that $\forall n, x,\ f_n(x) \to f(x)$ then f is a measurable function.*

Theorem 14. *Every continuous function $g : \mathbb{R} \to \mathbb{R}$ is measurable with respect to $(\mathcal{B}(\mathbb{R}), \mathcal{B}(\mathbb{R}))$.*

Theorem 15. *If $g : \mathbb{R} \to \mathbb{R}$ is continuous and f is measurable then $g \circ f$ is also measurable.*

Theorem 14 is a direct result of Theorem 8 stating that the inverse image of an open set by a continuous function is open. Theorem 15 guarantees, for instance, that if f is measurable then $exp(f)$, $Log(f)$, $cos(f)$ are measurable. This is derived using Theorem 14 and the equality $(g \circ f)^{-1}(A) = f^{-1}(g^{-1}(A))$. We now show how to prove that the sum of two measurable functions is measurable.

Proof. We need to prove that for any $c \in \mathbb{R}$, $(f+g)^{-1}(]-\infty, c[)$ is a measurable set. One way to solve this is to write it as a countable union of measurable sets. By definition of the inverse image, $(f+g)^{-1}(]-\infty, c[) = \{x : f(x) + g(x) < c\} = \{x : f(x) < c - g(x)\}$. Using Theorem 3 we prove that it is equal to $\bigcup_{r\in\mathbb{Q}}\{x : f(x) < r \text{ and } r < c - g(x)\}$. We deduce that $(f+g)^{-1}(]-\infty, c[) = \bigcup_{r\in\mathbb{Q}} f^{-1}(]-\infty, r[) \cap g^{-1}(]-\infty, c-r[)$. The right hand side is a countable union of measurable sets because $\mathbb{Q}$ is countable and f and g are measurable functions. □

5 Lebesgue Integral

Similar to the way in which step functions are used in the development of the Riemann integral, the Lebesgue integral makes use of a special class of functions called positive simple functions. They are measurable functions taking finitely many values. In other words, a positive simple function g can be written as a finite linear combination of indicator functions of measurable sets (a_i).

$$\forall x \in X,\ g(x) = \sum_{i\in s} \alpha_i I_{a_i}(x) \quad c_i \geq 0 \tag{1}$$

Let $(X, \mathcal{A}, \mu)$ be a measure space. The integral of the positive simple function g with respect to the measure μ is given by

$$\int_X g\, d\mu = \sum_{i\in s} \alpha_i \mu(a_i) \tag{2}$$

Various properties of the Lebesgue integral for positive simple functions have been proven in HOL [3]. We mention in particular that the integral above is well-defined and is independent of the choice of $(\alpha_i), (a_i), s$. Other properties include the linearity and monotonicity of the integral for positive simple functions. Another theorem that was widely used in [3] has however a serious constraint, as was discussed in the related work, where the author had to assume that every subset of the space X is measurable. Utilizing our formalization of the Borel algebra, we have been able to overcome this problem. The new theorem can be stated as

Theorem 16. *Let $(X, \mathcal{A}, \mu)$ be a measure space, f a non-negative function measurable with respect to $(\mathcal{A}, \mathcal{B}(\mathbb{R}))$ and (f_n) a monotonically increasing sequence of positive simple functions, pointwise convergent to f such that $\forall\, n, x,\ f_n(x) \leq f(x)$ then $\int_X f\, d\mu = lim_{n\to\infty} \int_X f_n\, d\mu$.*

The next step towards the Lebesgue integration for arbitrary measurable functions is the definition of the Lebesgue integral of positive measurable functions which is given by

$$\int_X f\, d\mu = \sup\{\int_X g\, d\mu \mid g \leq f \text{ and } g \text{ positive simple function}\} \tag{3}$$

Finally, the integral for arbitrary measurable functions is given by

$$\int_X f\, d\mu = \int_X f^+\, d\mu - \int_X f^-\, d\mu \tag{4}$$

Where f^+ and f^- are the positive functions defined by $f^+(x) = \max(f(x), 0)$ and $f^-(x) = \max(-f(x), 0)$.

5.1 Integrability

In this section, we provide the criteria of integrability of a measurable function and prove the integrability theorem which will play an important role in proving the properties of the Lebesgue integral.

Definition 7. *Let $(X, \mathcal{A}, \mu)$ be a measure space, a measurable function f is integrable iff $\int_X |f|\, d\mu < \infty$ or equivalently iff $\int_X f^+\, d\mu < \infty$ and $\int_X f^-\, d\mu < \infty$*

Theorem 17. *For any non-negative integrable function f there exists a sequence of positive simple functions (f_n) such that $\forall\, n, x,\ f_n(x) \leq f_{n+1}(x) \leq f(x)$ and $\forall\, x,\ f_n(x) \to f(x)$. Besides*

$$\int_X f\, d\mu = \lim_n \int_X f_n\, d\mu$$

For arbitrary integrable functions, the theorem is applied to f^+ and f^- and results in a well-defined integral, given by

$$\int_X f\, d\mu = \lim_n \int_X f_n^+\, d\mu - \lim_n \int_X f_n^-\, d\mu$$

Proof. Let the sequence (f_n) be defined as

$$f_n(x) = \sum_{k=0}^{4^n-1} \frac{k}{2^n} I_{\{x:\frac{k}{2^n} \leq f(x) < \frac{k+1}{2^n}\}} + 2^n I_{\{x:2^n \leq f(x)\}} \tag{5}$$

We show that the sequence (f_n) satisfies the conditions of the theorem and use Theorem 16 to conclude that $\int_X f\, d\mu = \lim_n \int_X f_n\, d\mu$. First, we use the definition of (f_n) to prove in HOL the following lemmas

Lemma 1. $\forall\, n, x,\ f(x) \geq 2^n \Rightarrow f_n(x) = 2^n$

Lemma 2. $\forall\, n, x,$ *and* $k < 4^n,\ \frac{k}{2^n} \leq f(x) < \frac{k+1}{2^n} \Rightarrow f_n(x) = \frac{k}{2^n}$

Lemma 3. $\forall\, x,\ (f(x) \geq 2^n) \vee (\exists k,\ k < 4^n$ *and* $\frac{k}{2^n} \leq f(x) < \frac{k+1}{2^n})$

Using these lemmas we prove that the sequence (f_n) is pointwise convergent to f $(\forall\, x,\ f_n(x) \to f(x))$, upper bounded by f $(\forall\, n, x,\ f_n(x) \leq f(x))$ and monotonically increasing $(\forall\, n, x,\ f_n(x) \leq f_{n+1}(x))$. □

5.2 Lebesgue Integral Properties

We prove in HOL key properties of the Lebesgue integral, in particular the monotonicity and linearity. Let f and g be integrable functions and $c \in \mathbb{R}$ then

Theorem 18. $\forall x,\ 0 \leq f(x) \Rightarrow 0 \leq \int_X f\, d\mu$

Theorem 19. $\forall x,\ f(x) \leq g(x) \Rightarrow \int_X f\, d\mu \leq \int_X g\, d\mu$

Theorem 20. $\int_X cf\, d\mu = c \int_X f\, d\mu$

Theorem 21. $\int_X f + g\, d\mu = \int_X f\, d\mu + \int_X g\, d\mu$

Theorem 22. *A and B disjoint sets* $\Rightarrow \int_{A\cup B} f\, d\mu = \int_A f\, d\mu + \int_B f\, d\mu$

Proof. We only show the proof for Theorem 21. We start by proving the property for non-negative functions. Using the integrability property, given in Theorem 17, there exists two sequences (f_n) and (g_n) that are pointwise convergent to f and g, respectively, such that $\int_X f\, d\mu = \lim_n \int_X f_n\, d\mu$ and $\int_X g\, d\mu = \lim_n \int_X g_n\, d\mu$. Let $h_n = f_n + g_n$ then the sequence h_n is monotonically increasing, pointwise convergent to $f + g$ and $\forall x\ h_n(x) \leq (f + g)(x)$ and using Theorem 16, $\int_X f + g\, d\mu = \lim_n \int_X h_n\, d\mu$. Finally, using the linearity of the integral for positive simple functions and the linearity of the limit, $\int_X f + g\, d\mu = \lim_n \int_X f_n\, d\mu + \lim_n \int_X g_n\, d\mu = \int_X f\, d\mu + \int_X g\, d\mu$. Now we consider arbitrary integrable functions. We first prove in HOL the following lemma.

Lemma 4. *If f_1 and f_2 are positive integrable functions such that $f = f_1 - f_2$ then $\int_X f\, d\mu = \int_X f_1\, d\mu - \int_X f_2\, d\mu$*

The definition of the integral is a special case of this lemma where $f_1 = f^+$ and $f_2 = f^-$. Going back to our proof, let $f_1 = f^+ + g^+$ and $f_2 = f^- + g^-$ then f_1 and f_2 are non-negative integrable functions satisfying $f + g = f_1 - f_2$. Using the lemma we conclude that
$\int_X f+g\, d\mu = \int_X f_1\, d\mu - \int_X f_2\, d\mu = (\int_X f^+\, d\mu + \int_X g^+\, d\mu) - (\int_X f^+\, d\mu + \int_X g^+\, d\mu) =$
$(\int_X f^+\, d\mu - \int_X f^-\, d\mu) + (\int_X g^+\, d\mu - \int_X g^-\, d\mu) = \int_X f\, d\mu + \int_X g\, d\mu.$ □

5.3 Lebesgue Monotone Convergence

The monotone convergence is arguably the most important theorem of the Lebesgue integration theory and it plays a major role in the proof of the Radon Nikodym theorem [2]. In this section, we present a proof of the theorem in HOL.

Theorem 23. *Let f be an integrable function and (f_n) be a sequence of functions such that $\forall n, x,\ 0 \leq f_n(x) \leq f_{n+1}(x) \leq f(x)$ and $\forall x,\ f_n(x) \to f(x)$. Then*

$$\int_X f\, d\mu = \lim_{n\to\infty} \int_X f_n\, d\mu$$

Proof. By the monotonicity of the integral, we deduce that $\forall\, n,\ \int_X f_n\, d\mu \leq \int_X f\, d\mu$. Hence $\lim_{n\to\infty} \int_X f_n\, d\mu \leq \int_X f\, d\mu$. It remains to prove that $\int_X f\, d\mu \leq \lim_{n\to\infty} \int_X f_n\, d\mu$. From Theorem 17, there exists a sequence of positive simple functions (g_n) such that $\forall\, n, x,\ g_n(x) \leq g_{n+1}(x) \leq f(x)$ and $\forall\, x,\ g_n(x) \to f(x)$ satisfying $\int_X f\, d\mu = \lim_{n\to\infty} \int_X g_n\, d\mu$. It is sufficient to prove that $\forall k \in \mathbb{N}$, $\int_X g_k\, d\mu \leq \lim_{n\to\infty} \int_X f_n\, d\mu$. For a fixed k, since g_k is a positive simple function then there exists $(\alpha_i), (a_i)$ and a finite set s such that $\int_X g_k\, d\mu = \sum_{i\in s} \alpha_i \mu(a_i)$. On the other hand, splitting the integral of f_n and using the properties of the integral and limit, we have $\lim_{n\to\infty} \int_X f_n\, d\mu = \lim_{n\to\infty} \sum_{i\in s} \int_X f_n I_{a_i}\, d\mu = \sum_{i\in s} \lim_{n\to\infty} \int_X f_n I_{a_i}\, d\mu$. Consequently, it suffices to prove that $\forall\, i \in s$, $\alpha_i \mu(a_i) \leq \lim_{n\to\infty} \int_X f_n I_{a_i}\, d\mu$ Or, equivalently, that $\forall\, i \in s$ and z such that $0 < z < 1$, $z\alpha_i \mu(a_i) \leq \lim_{n\to\infty} \int_X f_n I_{a_i}\, d\mu$. Let $b_n = \{t \in a_i : z\alpha_i \leq f_n(t)\}$ then $\bigcup_n b_n = a_i$ and $z\alpha_i \mu(a_i) = z\alpha_i \mu(\bigcup_n b_n) = z\alpha_i \lim_n \mu(b_n) = \lim_n z\alpha_i \mu(b_n) = \lim_n \int_X z\alpha_i I_{b_n}\, d\mu$. Furthermore, from the definition of b_n and the monotonicity of the integral, $\int_X z\alpha_i I_{b_n}\, d\mu \leq \int_X f_n I_{b_n}\, d\mu \leq \int_X f_n I_{a_i}\, d\mu$. We conclude that $z\alpha_i \mu(a_i) \leq \lim_{n\to\infty} \int_X f_n I_{a_i}\, d\mu$. □

5.4 Almost Everywhere

In this section we will define the "almost everywhere" relation [1] and prove in HOL some properties of the Lebesgue integral based on this relation. Consider a measure space $(X, \mathcal{A}, \mu)$. A null set E is a measurable set of measure zero.

Definition 8. *Almost Everywhere*
Let A be a subset of X and P be a property about elements of A. We say that P is true almost everywhere in A, abbreviated as "P a.e. in A", relative to the measure μ, if the subset of A where the property does not hold is a null set.

```
⊢ ∀m P. ae m P =
    {x | x ∈ m_space m ∧ ~P x} ∈ measurable_sets m ∧
    (measure m {x | x ∈ m_space m ∧ ~P x} = 0)
```

When $A = X$, we simply say "P a.e.". For example, $f = g$ a.e. means that the set $\{x \mid f(x) \neq g(x)\}$ is a null set.
Similarly, $f_n \to f$ a.e. means that there exists a null set E such that $\forall\, x \in X \backslash E \quad f_n(x) \to f(x)$.

Theorem 24. *If A is a null set then for any measurable function f, $\int_A f\, d\mu = 0$*

Theorem 25. *If f and g are two integrable functions such that $f = g$ almost everywhere, then $\int_X f\, d\mu = \int_X g\, d\mu$*

Theorem 26. *If f and g are two integrable functions such that $f \leq g$ almost everywhere, then $\int_X f\, d\mu \leq \int_X g\, d\mu$*

We provide the proof of the first theorem as it is used to prove the last two.

Proof. It suffices to prove the theorem for positive measurable functions as the integral of an arbitrary function g is the difference of the integrals of g^+ and g^-. By definition, $\int_A f\,d\mu = \int_X fI_A\,d\mu = \sup\{\int_X g\,d\mu \mid g \leq fI_A\}$ where the functions g are positive simple functions.
We will show that the set over which the supremum is taken is equal to $\{0\}$. For a positive simple function g such that $g \leq fI_A$ we show that $g(x) = 0$ outside of A. Hence $\int_X g\,d\mu = \int_A g\,d\mu = \int_X gI_A\,d\mu$. Furthermore, there exists $(\alpha_i), (a_i)$ and a finite set s such that $\forall x \in X,\ g(x) = \sum_{i\in s} \alpha_i I_{a_i}(x)$. The indicator function of A can be split as $I_A = \sum_{i\in s} I_{A\cap a_i}$. Hence gI_A can be written as $gI_A = \sum_{i\in s} \alpha_i I_{A\cap a_i}$. This implies that $\int_X gI_A\,d\mu = \sum_{i\in s} \alpha_i \mu(A \cap a_i)$. Since $0 \leq \mu(A \cap a_i) \leq \mu(A) = 0$ and s is finite, then $\int_X g\,d\mu = 0$ □

6 Applications

In this section, we use our formalized Lebesgue integration theory to prove in HOL some important properties from the theory of probability, namely, the Chebyshev and Markov inequalities and the Weak Law of Large Numbers [14].

6.1 Chebyshev and Markov Inequalities

In probability theory, both the Chebyshev and Markov inequalities provide estimates of tail probabilities. The Chebyshev inequality guarantees, for any probability distribution, that nearly all the values are close to the mean and it plays a major role in the derivation of the laws of large numbers [14]. The Markov inequality provides loose yet useful bounds for the cumulative distribution function of a random variable.

Let X be a random variable with expected value m and finite variance σ^2. The Chebyshev inequality states that for any real number $k > 0$,

$$P(|X - m| \geq k\sigma) \leq \frac{1}{k^2} \tag{6}$$

The Markov inequality states that for any real number $k > 0$,

$$P(|X| \geq k) \leq \frac{m}{k} \tag{7}$$

Instead of proving directly these inequalities, we provide a more general proof using measure theory and Lebesgue integrals in HOL that can be used for both and a number of similar inequalities. The probabilistic statement follows by considering a space of measure 1.

Theorem 27. *Let $(S, \mathcal{S}, \mu)$ be a measure space, and let f be a measurable function defined on S. Then for any nonnegative function g, nondecreasing on the range of f,*

$$\mu(\{x \in S : f(x) \geq t\}) \leq \frac{1}{g(t)} \int_S g \circ f\,d\mu\,.$$

```
⊢ ∀m f g t.
    (let A = {x | x ∈ m_space m ∧ t ≤ f x} in
       measure_space m ∧
       f ∈ measurable (m_space m,measurable_sets m) Borel ∧
       (∀x. 0 ≤ g x) ∧ (∀x y. x ≤ y ⇒ g x ≤ g y) ∧
       integrable m (\x. g (f x)) ⇒
       measure m A ≤ (1 / (g t)) * fn_integral m (\x. g (f x)))
```

The Chebyshev inequality is derived by letting $t = k\sigma$, $f = |X - m|$ and g defined as $g(t) = t^2$ if $t \geq 0$ and 0 otherwise. The Markov inequality is derived by letting $t = k$, $f = |X|$ and and g defined as $g(t) = t^2$ if $t \geq 0$ and 0 otherwise.

Proof. Let $A = \{x \in S : t \leq f(x)\}$ and I_A be the indicator function of A. From the definition of A, $\forall x\ 0 \leq g(t)I_A(x)$ and $\forall x \in A\ t \leq f(x)$. Since g is non-decreasing, $\forall x,\ g(t)I_A(x) \leq g(f(x))I_A(x) \leq g(f(x))$. As a result, $\forall x\ g(t)I_A(x) \leq g(f(x))$. A is measurable because f is $(\mathcal{S}, \mathcal{B}(\mathbb{R}))$ measurable. Using the monotonicity of the integral, $\int_S g(t)I_A(x)d\mu \leq \int_S g(f(x))d\mu$. Finally from the linearity of the integral $g(t)\mu(A) \leq \int_S g \circ f d\mu$. □

6.2 Weak Law of Large Numbers (WLLN)

The WLLN states that the average of a large number of independent measurements of a random quantity converges in probability towards the theoretical average of that quantity. Interpreting this result, the WLLN states that for a sufficiently large sample, there will be a very high probability that the average will be close to the expected value. This law is used in a multitude of fields. It is used, for instance, to prove the asymptotic equipartition property [4], a fundamental concept in the field of information theory.

Theorem 28. *Let $X_1, X_2, ...$ be an infinite sequence of independent, identically distributed random variables with finite expected value $E[X_1] = E[X_2] = ... = m$ and let $\overline{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$ then for any $\varepsilon > 0$,*

$$\lim_{n \to \infty} P(|\overline{X} - m| < \varepsilon) = 1 \tag{8}$$

```
⊢ ∀p X m v e.
  prob_space p ∧ 0 < e ∧
  (∀i j. i ≠ j ⇒ uncorrelated p (X i) (X j)) ∧
  (∀i. expectation p (X i) = m) ∧ (∀i. variance p (X i) = v) ⇒
    lim (\n. prob p {x | x ∈ p_space p ∧
      abs ((\x. 1/n * SIGMA (\i. X i x) (count n))x-m) < e}) = 1
```

Besides the Chebyshev inequality, to prove this theorem in HOL, we need to formalize and prove some key properties of the variance of a random variable. The main property being that the variance of a sum of uncorrelated random variables is the sum of their variances. Notice that the requirement of the random variables being independent in the WLLN can be relaxed to simply requiring them to be uncorrelated.

Let X and Y be random variables with expected values μ_X and μ_Y, respectively. The variance of X is given by $Var(X) = E[|X - \mu_X|^2]$ and the covariance between X and Y is given by $Cov(X, Y) = E[(X - \mu_X)(Y - \mu_Y)]$. X and Y are uncorrelated iff $Cov(X, Y) = 0$.

We prove the following properties in HOL: $Var(X) = E[X^2] - \mu_X^2$; $Cov(X, Y) = E[XY] - \mu_X\mu_Y$; $Var(X) \geq 0$ and $Var(aX) = a^2Var(X)$. $Var(X + Y) = Var(X) + Var(Y) + 2Cov(X, Y)$ and $Var(X + Y) = Var(X) + Var(Y)$ if X and Y are uncorrelated. Finally if $\forall i \neq j, X_i$, X_j are uncorrelated, then $Var(\sum_{i=1}^{N} X_i) = \sum_{i=1}^{N} Var(X_i)$.

Proof. Using the linearity property of the Lebesgue integral as well as the properties of the variance we prove that $E[\overline{X}] = \frac{1}{N}\sum_{i=1}^{N} m = m$ and $Var(\overline{X}) = \frac{\sigma^2}{N}$. Applying the Chebyshev inequality to $\overline{X}$, we get $P(|\overline{X} - m| \geq \varepsilon) \leq \frac{\sigma^2}{N\varepsilon^2}$. Equivalently, $1 - \frac{\sigma^2}{N\varepsilon^2} \leq P(|\overline{X} - m| < \varepsilon) \leq 1$. It then follows that $\lim_{n\to\infty} P(|\overline{X} - m| < \varepsilon) = 1$. □

To prove the results of this section in HOL we used the Lebesgue integral properties, in particular, the monotonicity and the linearity, as well as the properties of real-valued measurable functions. The above is not available in the work of Coble [3] because his formalization does not include the Borel sets so he cannot prove the Lebesgue properties and the theorems of this section. The Markov and Chebyshev inequalities were previously proven by Hasan and Tahar [9] but only for discrete random variables. Our formalization allows us to provide a proof valid for both the discrete and continuous cases. Richter's formalization [15] only allows random variables defined on the whole universe of a certain type. The above mentioned formalizations do not include the definition of variance and proofs of its properties and hence cannot be used to verify the WLLN.

7 Conclusions

In this paper, we have presented a formalization in HOL of the Borel algebra to fill the gap in previous formalizations in higher-order-logic of the Lebesgue integral. Our formalization is general as it can be applied on functions defined on any metric space. Building on this framework, we proved important properties of the Lebesgue integral, in particular, the monotonicity and linearity properties. We also proved in HOL the Lebesgue monotone convergence, a key result of the Lebesgue integration theory. Additionally, we formalized the concept of "almost everywhere" and proved that the Lebesgue integral does not distinguish between functions which differ on a null set as well as other important results based on the "almost everywhere" relation. These features of the proposed approach facilitate the formal reasoning process for the continuous and unpredictable components of a wide range of physical systems. For illustration purposes, we proved in HOL key theorems from the theory of probability, namely the Chebyshev and Markov inequalities as well as the WLLN. The HOL codes corresponding to all the formalization and proofs, presented in this paper, are available in [12].

Overall our formalization required more than 7000 lines of code. Only 250 lines were required to verify the key properties of the applications section. This shows the significance of our work in terms of simplifying the formal proof of properties using the Lebesgue integration theory. The main difficulties encountered were the multidisciplinary nature of this work, requiring deep knowledge of measure and integration theories, topology, set theory, real analysis and probability and information theory. Some of the mathematical proofs also posed challenges to be implemented in HOL.

Our future plans include using the Lebesgue integral development to formalize key concepts of the information theory. We will use the Lebesgue monotone convergence theorem and the Lebesgue integral properties to prove the Radon Nikodym theorem [2], paving the way to defining the probability density functions as well as the Kullback-Leibler divergence [4], which is related to the mutual information, entropy and conditional entropy [4].

References

1. Berberian, S.K.: Fundamentals of Real Analysis. Springer, Heidelberg (1998)
2. Bogachev, V.I.: Measure Theory. Springer, Heidelberg (2006)
3. Coble, A.R.: Anonymity, Information, and Machine-Assisted Proof. PhD thesis, University of Cambridge, UK (2009)
4. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley-Interscience, Hoboken (1991)
5. Goldberg, R.R.: Methods of Real Analysis. Wiley, Chichester (1976)
6. Halmos, P.R.: The foundations of probability. The American Mathematical Monthly 51(9), 493–510 (1944)
7. Harrison, J.: Theorem Proving with the Real Numbers. Springer, Heidelberg (1998)
8. Harrison, J.: A HOL Theory of Euclidean Space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005)
9. Hasan, A., Tahar, S.: Formal Verification of Tail Distribution Bounds in the HOL Theorem Prover. Mathematical Methods in the Applied Sciences 32(4), 480–504 (2009)
10. Hurd, J.: Formal Verifcation of Probabilistic Algorithms. PhD thesis, University of Cambridge, UK (2002)
11. Lester, D.: Topology in PVS: Continuous Mathematics with Applications. In: workshop on Automated Formal Methods, pp. 11–20. ACM, New York (2007)
12. Mhamdi, T., Hasan, O., Tahar, S.: Formalization of the Lebesgue Integration Theory in HOL. Technical Report, ECE Dept., Concordia University (April 2009), http://hvg.ece.concordia.ca/Publications/TECH_REP/MLP_TR10/
13. Munkres, J.: Topology. Prentice Hall, Englewood Cliffs (1999)
14. Papoulis, A.: Probability, Random Variables, and Stochastic Processes. Mc-Graw Hill, New York (1984)
15. Richter, S.: Formalizing Integration Theory with an Application to Probabilistic Algorithms. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 271–286. Springer, Heidelberg (2004)

From Total Store Order to Sequential Consistency: A Practical Reduction Theorem*

Ernie Cohen[1] and Bert Schirmer[2]

[1] European Microsoft Innovation Center, Aachen, Germany
[2] German Research Center for Artificial Intelligence (DFKI) Saarbrücken, Germany
ernie.cohen@microsoft.com, norbert.schirmer@dfki.de

Abstract. When verifying a concurrent program, it is usual to assume sequentially consistent memory. However, most modern multiprocessors buffer their stores, providing native sequential consistency only at a substantial performance penalty. To regain sequential consistency, a programmer has to follow an appropriate programming discipline. However, existing naïve disciplines, such as protecting all shared accesses with locks to avoid data races, or flushing store buffers according to a protocol that allows arbitrary data races, are not flexible enough for building high-performance multiprocessor software. We present a new discipline for concurrent programming under TSO (total store order, with store buffer forwarding). Instead of using concurrency primitives, such as locks, it is based on maintaining ownership information in ghost state, allowing the discipline to be expressed as a state invariant and verified through conventional program reasoning. If every execution of a program in a system without store buffers follows the discipline, then every execution of the program in a system with store buffers is sequentially consistent.

1 Introduction

When verifying a shared-memory concurrent program, it is usual to assume that each memory operation works directly on a shared memory state, a model sometimes called *atomic* memory. A memory implementation that provides this abstraction for programs that communicate only through shared memory is said to be *sequentially consistent*. Concurrent algorithms in the computing literature tacitly assume sequential consistency, as do most application programmers.

However, modern computing platforms do not guarantee sequential consistency for arbitrary programs, for two reasons. First, optimizing compilers are typically incorrect unless the program is appropriately annotated to indicate which program locations might be concurrently accessed by other threads; this issue is addressed only cursorily in this paper. Second, modern processors buffer stores of retired instructions. To make such buffering transparent to single-processor programs, subsequent reads of the processor read from the buffer in preference

* Work funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 403–418, 2010.

to the cache (store buffer forwarding); otherwise, a program could write a new value to an address but later read an older value. However, in a multiprocessor system, processors do not snoop the store buffers of other processors, so a store becomes visible to the storing processor before it becomes visible to others. This can result in executions that are not sequentially consistent.

The simplest example illustrating such an inconsistency is the following program, consisting of two threads T0 and T1, where `x` and `y` are shared memory variables (initially 0) and `r0` and `r1` are registers:

```
T0: x = 1;        T1: y = 1;
    r0 = y;           r1 = x;
```

In a sequentially consistent execution, it is impossible for both `r0` and `r1` to be assigned 0. This is because the assignments to `x` and `y` must be executed in some order; if `x` (resp. `y`) is assigned first, then `r1` (resp. `r0`) will be set to 1. However, in the presence of store buffers, the assignments to `r0` and `r1` might be performed while the writes to `x` and `y` are still in their respective store buffers, resulting in both `r0` and `r1` being assigned 0.

One way to cope with store buffers is to make them an explicit part of the programming model which is a substantial programming concession. Because store buffers are FIFO this ratchets up the complexity of program reasoning considerably. Moreover, because writes from function calls might still be buffered when a function returns, visible store buffers break modular program reasoning.

In practice, the usual remedy for store buffering is adherence to a programming discipline that provides sequential consistency for a suitable class of architectures. In this paper, we describe and prove the correctness of such a discipline suitable for the memory model provided by existing x86/x64 machines, where each write emerging from a store buffer hits a global memory subsystem visible to all processors. Because each processor sees the same global ordering of writes, this model is sometimes called *total store order* (TSO) [2][1]

The concurrency discipline most familiar to concurrent programs is one where each variable is protected by a lock, and a thread must hold the corresponding lock to access the variable. (It is possible to generalize this to allow shared locks, as well as variants such as split semaphores.) Such lock-based techniques are typically referred to as *coarse-grained* concurrency control, and suffice for most concurrent application programming. However, these techniques do not suffice for low-level system programming (e.g., the construction of OS kernels), for several reasons. First, in kernel programming efficiency is paramount, and atomic memory operations are more efficient for many problems. Second, lock-free concurrency control can sometimes guarantee stronger correctness (e.g., wait-free algorithms can provide bounds on execution time). Third, kernel programming

[1] In 2008, both Intel [8] and AMD [1] put forward a weaker memory model in which writes to different memory addresses might be seen in different orders on different processors, but must respect causal ordering. However, current implementations satisfy the stronger conditions described in this paper, and Intel has backed away from these proposals, reaffirming TSO compliance in its latest specifications [9]. According to Owens et al. [12] AMD plans a similar adaptation of their manuals.

requires taking into account the implicit concurrency of concurrent hardware activities (e.g., a hardware TLB racing to use page tables while the kernel is trying to access them), and hardware cannot be forced to follow a locking discipline.

A more refined concurrency control discipline, which is much closer to expert practice, is to classify memory addresses as lock-protected or shared. Lock-protected addresses are used in the usual way, but shared addresses can be accessed using atomic operations provided by hardware (e.g., on x86 class architectures, most reads and writes are atomic[2]). The main disciplinary restriction on these accesses is that if a processor does a shared write and a subsequent shared read (possibly from a different address), the processor must flush the store buffer somewhere in between. A flush is triggered by a memory *fence* or *interlocked* instructions (like compare and swap). In our example, both `x` and `y` would be shared addresses, so each processor would have to flush its store buffer between its first and second operations. The C-idiom to identify shared portions of memory is the `volatile` tag on variables and type declarations. Temporarily thread local data (like lock-protected addresses) can be accessed non-volatilely, whereas accesses to shared memory are tagged as volatile. This prevents a compiler from applying certain optimizations to shared accesses which could cause undesired behavior, e.g., to store intermediate values in registers instead of writing them to the memory.

However, even this discipline is not very satisfactory. First, we would need more rules to allow locks to be created or destroyed, or to change memory between shared and protected, and so on. Second, there are many interesting concurrency control mechanisms besides locks that allow a thread to obtain exclusive or shared access to data, and this discipline would not help with these.

In this paper, we consider a much more general and powerful discipline that also guarantees sequential consistency. The basic rule for shared addresses is similar to the discipline above, but there are no locking primitives. Instead, we treat *ownership* as fundamental; local accesses (i.e., operations on locations owned by the thread) do not introduce any flushing obligations. Using ownership as a basis of our reduction has a number of advantages:

1. Ownership is manipulated by nonblocking ghost updates, whereas locking (or other concurrency control primitives) introduce runtime overheads.
2. Instead of building concurrency control primitives (such as locks) into the methodology, we can instead verify that they conform to our discipline.
3. Because our discipline can be expressed as an ordinary program invariant, conformance to the discipline can be established using ordinary (sequentially consistent) concurrent program reasoning, without having to talk about histories or complete executions.
4. Important compiler optimizations (e.g., moving a local write ahead of a global read) require distinguishing between owned and unowned accesses, so this is already built into important low-level languages (such as C).

Overview. In Sect. 2 we describe the programming discipline. In Sect. 3 we introduce preliminaries of Isabelle/HOL, the theorem prover in which we mechanized

[2] Except for certain memory types, or for operations that cross a cache line.

our work,[3] which is detailed in Sect. 4 introducing the formal models and the reduction theorem. We conclude and discuss related work in Sect. 5.

2 Programming Discipline

The basic idea behind the programming discipline is, that before gathering new information about the shared (volatile) state (via reading) the thread has to make its outstanding changes to the shared state (volatile writes) visible to others, by flushing the store buffer. Outstanding changes to thread local memory (non-volatile writes) do not require flushing. This makes it possible to construct a sequentially consistent execution of the global system. In this execution, a volatile write to shared memory happens when the write instruction exits the store buffer, and a volatile read from the shared memory happens when all preceding writes have exited the store buffer. We distinguish thread local and shared memory by an ownership model. Ownership is maintained in ghost state and can be transferred either as side effect of a write operation or by a dedicated ghost operation. Informally, the rules of the discipline are as follows:

- In any state, each memory address is either (i) *shared* or *unshared*, (ii) *unowned* or *owned* by a unique thread, and (iii) *read-only* or *read-write*.
- Every unowned address must be shared and can be accessed by all threads.
- Every read-only address is unowned.
- A thread can take ownership of an unowned address, or release ownership of an address that it owns. It can also change whether an address it owns is shared or not. Upon release of an address it can mark it as read-only.
- Each memory access is marked as *volatile* or *non-volatile*.
- Every thread maintains a *dirty* flag, which is set on a volatile write and *cleaned* on every flushing instruction (i.e. fence and interlocked instructions).
- A thread can only perform *sound* accesses:
  - a non-volatile write if the address is owned by the thread and is unshared,
  - a non-volatile read if the address is owned by the thread or is read-only,
  - a volatile read if the address is shared or the thread owns it; the dirty flag of the thread has to be clean,
  - a volatile write if no other thread owns the address (i.e. unowned or owned by the writer) and the address is not read-only.
  - For interlocked instructions, which have the side effect of the store buffer getting flushed, the rules for volatile accesses apply.

Note first that the conditions above are not thread-local, because some actions are allowed only when an address is unowned, marked read-only, or not marked read-only. A thread can ascertain such conditions only through system-wide invariants, respected by all threads, along with data it reads. By imposing suitable global invariants, various thread-local disciplines (such as one where addresses are protected by locks, conditional critical regions, or monitors) can be derived as lemmas by ordinary program reasoning, without need for meta-theory.

[3] Theory files will be published on the AFP: `http://afp.sourceforge.net/`

Table 1. Programming discipline

(a) Access policy

	sharing		
owner	read-write	read-only	unshared
unowned	vR, vW	nvR	/
me	nvR, vW	/	nvR, nvW
other	vR	/	/
volatile reads have to be clean			

(b) Flushing policy

	flush (before)
interlocked	as side effect
vR	if dirty
others	never

(**v**olatile, **n**on-**v**olatile, **R**ead, **W**rite)

Second, note that these rules can be checked in the context of a concurrent program without store buffers, by introducing ghost state to keep track of ownership and sharing and the dirty flag to monitor whether the thread has performed a volatile write since the last flush. The dirty flag only has to be clean for volatile reads, for non-volatile reads it can be dirty. This means that only volatile reads may require store buffer flushes to ensure sequential consistency. Our main result is that if a program obeys the rules above, then the program is sequentially consistent when executed on a TSO machine.

Table 1a summarizes the access policy and Table 1b the associated flushing policy of the programming discipline. Typical use-cases are the following: unowned (read-write) memory is used for lock-free algorithms or the locks themselves; when acquiring a lock the protected data becomes owned and unshared; owned and shared memory captures the single-writer-multiple-readers idiom; read-only (unowned) memory is used for memory protected by a reader-writer lock while there is no writer, or if data stays unmodified after initialization. The key motivation is to improve performance by minimizing the number of store buffer flushes, while staying sequentially consistent. The need for flushing the store buffer decreases from interlocked accesses (where flushing is a side-effect) over volatile accesses to non-volatile accesses. Accepting the performance penalty it is sound to use stricter accesses as necessary e.g. use a volatile access if a non-volatile one is sufficient. The access rights of interlocked and volatile accesses coincide. Some interlocked operations (e.g. test-and-set) can read from, modify and write to an address in a single atomic hardware step.When a thread owns an address it is guaranteed that it is the only one writing to that address. This thread can safely perform non-volatile reads to that address without missing any write. Similar it is safe for any thread to access read-only memory via non-volatile reads since there are no outstanding writes at all.

Reconsider our first example program. If we choose to leave both `x` and `y` unowned (and hence shared), then all accesses must be volatile. This forces each thread to flush its store buffer between its write and read operation. We mark assignments and reads with subscripts v for volatile and nv for non-volatile accesses. Moreover, ghost operations are slanted and $\langle \dots \rangle$ make the enclosed operations happen atomically. We highlight some preconditions by inserting an

assertion $\{\ldots\}$. Note that the dirty flags are thread local and we can use ordinary sequential reasoning to argue that they remain reset after the fence operations.

```
T0: ⟨x_v = 1, dirty = true⟩;        T1: ⟨y_v = 1, dirty = true⟩;
    ⟨fence, dirty = false⟩;             ⟨fence, dirty = false⟩;
    ⟨{dirty == false}, r0 = y_v⟩;       ⟨{dirty == false}, r1 = x_v⟩;
```

In practice, on an x86/x64 machine, the fence may be omitted by making the writes interlocked, which flushes store buffers as a side effect. For whichever thread the write is the second to hit memory the subsequent read is guaranteed to see the write of the other thread, making the execution violating sequential consistency impossible.

However, couldn't the first thread take ownership of `x` before writing it, so that its write could be non-volatile? The answer is that it could try, but then the second thread would be unable to read `x` volatile (or take ownership of `x` and read it non-volatile), because we would be unable to prove that `x` is unowned at that point. In other words, a thread can take ownership of an address only if it is not racing to do so. This is illustrated by the following code snippet, where it is easy to construct an interleaving where an assertion is violated. Keep in mind that *acquire* and *release* are ghost operations which do not have any influence on possible schedules. We have to be able to proof that their preconditions always hold (for every possible schedule), e.g., that an address is unowned when we attempt to acquire it. In the examples after an *acquire(a)* the thread owns the address `a` and also marks it as unshared.

```
T0: ⟨{unowned(x)}, acquire(x)⟩;   T1: ⟨{unowned(y)}, acquire(y)⟩;
    x_nv = 1;                         y_nv = 1;
    release(x);                       release(y);
    ⟨{unowned(y)}, acquire(y)⟩;       ⟨{unowned(x)}, acquire(x)⟩;
    r0 = y_nv;                        r1 = x_nv;
    release(y);                       release(x);
```

The assertions do not hold for every possible schedule, as for example `x` may not be released by T0 at the point were T1 is trying to acquire it. Hence, the program does not obey our programming discipline.

Ultimately, the races allowed by the discipline involve volatile access to an unowned address, which brings us back to locks. A spinlock is typically implemented with an interlocked read-modify-write on an address, say `l` (the interlocking providing the required flushing of the store buffer). If the locking succeeds, we can prove that no other thread holds the lock, and can therefore safely take ownership of an address "protected" by the lock, say `a`. The global invariant to do this reasoning is that, whenever `l == 0` address `a` is unowned. And whenever `l == 1` there exists exactly one thread owning `a`. Thus, our discipline subsumes the better-known disciplines governing coarse-grained concurrency control. The following code snippet illustrates this reasoning in our framework:

```
while(!⟨interlocked_test_and_set(l_v)), acquire(a)⟩);
{'a owned by current thread'}
... critical section accessing a non-volatilely ...
⟨l_v = 0, release(a)⟩;
```

The *acquire(a)* is supposed to happen atomically with the successful test-and-set. It transfers ownership of the currently unowned a (according to the invariant) to the (thread-local) ownership-set of the current thread. Only the thread itself is able to release ownership once it has acquired it. This justifies non-volatile accesses to a in the critical section. Atomically with resetting the lock l the ownership of a is released, maintaining the invariant.

3 Preliminaries of Isabelle/HOL

The formalization presented in this paper is mechanized within the interactive theorem prover *Isabelle/HOL* [10], using its document generation facilities, which guarantees a close correspondence between the presentation and the theory files. We distinguish formal entities typographically from other text. We use a sans serif font for types and constants (including functions and predicates), e.g., map, a slanted serif font for free variables, e.g., x or ϑ, and a slanted sans serif font for bound variables, e.g., x or $\mathit{\theta}$. Small capitals are used for data type constructors, e.g., Foo, and type variables have a leading tick, e.g., $'a$. HOL keywords are typeset in type-writer font, e.g., `let`. The logical/mathematical notions follow the standard notational conventions with a bias towards functional programming. We prefer curried function application, e.g., $f\ a\ b$ instead of $f(a, b)$.

Isabelle/HOL provides standard types like Booleans, natural numbers, integers, total functions, pairs, lists, and sets. There are packages to define new data types and records. Isabelle allows polymorphic types, e.g., $'a$ list is the list type with type variable $'a$. In HOL all functions are total, e.g., nat $\Rightarrow$ nat is a total function on natural numbers. A function update is $f(y := v) \equiv \lambda x.$ `if` $x = y$ `then` v `else` $f\ x$. To formalize partial functions the type $'a$ option is used. It is a data type with two constructors, one to inject values of the base type, e.g., $\lfloor x \rfloor$, and the additional element $\bot$. A base value can be projected with the function the, which is defined by the sole equation the $\lfloor x \rfloor = x$. Since HOL is a total logic the term the $\bot$ is still a well-defined yet un(der)specified value. Partial functions are usually represented by the type $'a \Rightarrow 'b$ option.They are commonly used as *maps*. We denote the domain of map m by dom m. A map update is written as $m(a \mapsto v)$. We can restrict the domain of a map m to a set A by $m|_A$.

The empty list is $[]$, with $x \cdot xs$ the element x is 'consed' to the list xs.With xs @ ys list ys is appended to list xs. With the term map $f\ xs$ the function f is applied to all elements in xs. The length of a list is $|xs|$, the n-th element can be selected with $xs_{[n]}$ and is updated via $xs[n := v]$.

Sets come along with the standard operations like union, i.e., $A \cup B$, membership, i.e., $x \in A$ and set inversion, i.e., $-A$. Tuples with more than two components are pairs nested to the right.

4 Formalization

In this section we go into the details of our formalization. In our model, we distinguish the plain 'memory system' from the 'programming language semantics'

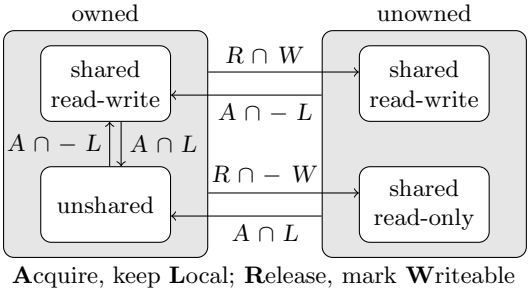


Fig. 1. Ownership transfer

which we both describe as a small-step transition relation. During a computation the programming language issues memory instructions (read / write) to the memory system, which itself returns the results in temporary registers. This interface allows us to parameterize the program semantics over the memory system. Our main theorem allows us to simulate a computation in the semantics based on a memory system with store buffers by a computation in the semantics based on a sequentially consistent memory system. We refer to the former one as *store buffer machine* and to the latter one as *virtual machine*. The simulation theorem is independent of the programming language. An instantiation with a simple WHILE language can be found in our technical report [5]. We continue with introducing the common parts of both machines.

Addresses a, values v and temporaries t are natural numbers. Ghost annotations for manipulating the ownership information are the following sets of addresses: the acquired addresses A, the unshared (local) fraction L of the acquired addresses, the released addresses R and the writable fraction W of the released addresses (the remaining addresses are considered read-only). These four annotations are part of write instructions, interlocked operations and a dedicated ghost instruction and are supplied by the verification engineer. According to these annotations ownership transfer can take place atomically with volatile writes and interlocked operations (in case a write is performed) or as sole effect of the ghost instruction. The possible status changes of an address due to these ownership transfer operations are depicted in Figure 1. For example by putting address a into the sets R and W a thread can release a, which it currently owns, and mark it as unowned and read-write. Otherwise, if $a \notin W$, its new state is unowned and read-only. Conversely an currently unowned address can be acquired by by putting it into set A. If it is also in set L, the new state is owned and unshared, otherwise it is owned and shared. The sharing state of an already owned address can also be altered accordingly. Note that ownership of an address is not directly transferred between threads, but is first released by one thread and then can be acquired by another thread.

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ \vartheta,\ sb) \qquad \vartheta \vdash p \rightarrow_{\mathsf{p}} (p',\ is')}{(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts[i := (p',\ is \mathbin{@} is',\ \vartheta,\ sb)],\ m)}$$

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ \vartheta,\ sb) \qquad (is,\ \vartheta,\ sb,\ m) \overset{\mathsf{sb}}{\rightarrow}_{\mathsf{m}} (is',\ \vartheta',\ sb',\ m')}{(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts[i := (p,\ is',\ \vartheta',\ sb')],\ m')}$$

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ \vartheta,\ sb) \qquad (m,\ sb) \rightarrow_{\mathsf{sb}} (m',\ sb')}{(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts[i := (p,\ is,\ \vartheta,\ sb')],\ m')}$$

Fig. 2. Global transitions of store buffer machine

A memory instruction is a datatype with the following constructors:

- Read *volatile a t* for reading from address *a* to temporary *t*, where the Boolean *volatile* determines whether the access is volatile or not.
- Write *volatile a sop A L R W* to write the result of evaluating the store operation *sop* at address *a*. A store operation is a pair $(D,\ f)$, with the domain D and the function f. The function f takes temporaries ϑ as a parameter, which maps a temporary to a value. The subset of temporaries that is considered by function f is specified by the domain D. We consider store operations as valid when they only depend on their domain:

  $$\mathsf{valid\text{-}sop}\ sop \equiv \forall D\ f\ \theta.\ sop = (D,\ f) \wedge D \subseteq \mathsf{dom}\ \theta \longrightarrow f\ \theta = f\ (\theta{\restriction_D})$$

  Again the Boolean *volatile* specifies the kind of memory access.
- RMW *a t sop cond ret A L R W*, for atomic interlocked 'read-modify-write' instructions (flushing the store buffer) which can affect both the temporaries and the memory. First the value at address *a* is loaded to temporary *t*, and then the condition *cond* on the temporaries is considered to decide whether a store operation *sop* is also executed. In case of a store the function *ret*, depending on both the old value at address *a* and the new value, specifies the final result stored in temporary *t*. With a trivial condition *cond* this instruction also covers interlocked reads and writes.
- Fence, a memory fence that flushes the store buffer.
- Ghost *A L R W* for ownership transfer.

4.1 Store Buffer Machine

The store buffer machine does not maintain any ghost state. A thread configuration is a tuple $(p,\ is,\ \vartheta,\ sb)$ consisting of the program state p, a memory instruction list is, the map of temporaries ϑ and the store buffer sb. A global configuration $(ts,\ m)$ consists of a thread list ts and the memory m, which is a function from addresses to values. Figure 2 defines the computation of the global system by the non-deterministic transition relation $(ts,\ m) \overset{\mathsf{sb}}{\Rightarrow} (ts',\ m')$. A transition selects a thread $ts_{[i]} = (p,\ is,\ \vartheta,\ sb)$ and either the 'program' the 'memory' or the 'store buffer' makes a step defined by sub-relations.

The program step relation $\vartheta \vdash p \rightarrow_{\mathsf{p}} (p',\ is')$ is an unspecifed parameter to the global transition relation. It takes temporaries ϑ and the current program state

$$\frac{v = (\mathsf{case\ buffered\text{-}val}\ sb\ a\ \mathsf{of}\ \bot \Rightarrow m\ a \mid \lfloor v' \rfloor \Rightarrow v')}{(\textsc{Read}\ volatile\ a\ t \cdot is,\ \vartheta,\ sb,\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is,\ \vartheta(t \mapsto v),\ sb,\ m)}$$

$$\frac{sb' = sb\ @\ [\textsc{Write}_{\mathsf{sb}}\ volatile\ a\ (D,\ f)\ (f\ \vartheta)\ A\ L\ R\ W]}{(\textsc{Write}\ volatile\ a\ (D,\ f)\ A\ L\ R\ W \cdot is,\ \vartheta,\ sb,\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is,\ \vartheta,\ sb',\ m)}$$

$$\frac{}{(m,\ \textsc{Write}_{\mathsf{sb}}\ volatile\ a\ sop\ v\ A\ L\ R\ W \cdot sb) \rightarrow_{\mathsf{sb}} (m(a := v),\ sb)}$$

$$\frac{\neg\ cond\ (\vartheta(t \mapsto m\ a)) \qquad \vartheta' = \vartheta(t \mapsto m\ a)}{(\text{RMW}\ a\ t\ (D,\ f)\ cond\ ret\ A\ L\ R\ W \cdot is,\ \vartheta,\ [],\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is,\ \vartheta',\ [],\ m)}$$

$$\frac{cond\ (\vartheta(t \mapsto m\ a)) \qquad \vartheta' = \vartheta(t \mapsto ret\ (m\ a)\ (f\ (\vartheta(t \mapsto m\ a)))) \qquad m' = m(a := f\ (\vartheta(t \mapsto m\ a)))}{(\text{RMW}\ a\ t\ (D,\ f)\ cond\ ret\ A\ L\ R\ W \cdot is,\ \vartheta,\ [],\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is,\ \vartheta',\ [],\ m')}$$

$$\frac{}{(\textsc{Ghost}\ A\ L\ R\ W \cdot is,\ \vartheta,\ sb,\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is,\ \vartheta,\ sb,\ m)}$$

$$\frac{}{(\textsc{Fence} \cdot is,\ \vartheta,\ [],\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is,\ \vartheta,\ [],\ m)}$$

Fig. 3. Memory and store buffer transitions of store buffer machine

p and makes a step by returning a new program state p' and an instruction list is' which is appended to the remaining instructions. For example executing an assignment `l = 0` to a volatile address l, with no further ownership annotations, generates the memory instruction $\textsc{Write}\ \mathsf{True}\ l\ (\emptyset,\ \lambda\theta.\ 0)\ \emptyset\ \emptyset\ \emptyset\ \emptyset$.

A memory step $(is,\ \vartheta,\ sb,\ m) \xrightarrow{\mathsf{sb}}_{\mathsf{m}} (is',\ \vartheta',\ sb',\ m')$ may only fill its store buffer with new writes. In a separate store buffer step $(m,\ sb) \rightarrow_{\mathsf{sb}} (m',\ sb')$ the store buffer may release outstanding writes to the memory.

The store buffer maintains the list of outstanding memory writes. Write instructions are appended to the end of the store buffer and emerge to memory from the front of the list. A read instruction is satisfied from the store buffer if possible. An entry in the store buffer is of the form $\textsc{Write}_{\mathsf{sb}}\ volatile\ a\ sop\ v$ for an outstanding write (keeping the volatile flag), where operation sop evaluated to value v. The memory and store buffer transitions are defined in Figure 3. For a read we obtain the value of the last write to address a which is still pending in the store buffer with $\mathsf{buffered\text{-}val}\ sb\ a$. In case no outstanding write to address a is in the store buffer we read the content of a from memory. Write operations have no immediate effect on the memory but are queued in the store buffer instead, they update the memory when they exit the store buffer. Interlocked and fence operations require an empty store buffer, which means that it has to be flushed before the action can take place. The read-modify-write instruction first adds the current value at address a to temporary t and then checks the store condition $cond$ on the temporaries. If it fails this read is the final result of the operation. Otherwise the store is performed. The resulting value of the temporary t is specified by the function ret which considers both the old and new value as input. The fence and the ghost instruction are just skipped.

$$\frac{i < |ts| \qquad ts_{[i]} = (p,\ is,\ \vartheta,\ \mathcal{D},\ \mathcal{O}) \qquad \vartheta \vdash p \rightarrow_{\mathsf{p}} (p',\ is')}{(ts,\ m,\ \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow} (ts[i := (p',\ is\ @\ is',\ \vartheta,\ \mathcal{D},\ \mathcal{O})],\ m,\ \mathcal{S})}$$

$$\frac{\begin{array}{c} i < |ts| \\ ts_{[i]} = (p,\ is,\ \vartheta,\ \mathcal{D},\ \mathcal{O}) \qquad (is,\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S}) \overset{\mathsf{v}}{\rightarrow}_{\mathsf{m}} (is',\ \vartheta',\ m',\ \mathcal{D}',\ \mathcal{O}',\ \mathcal{S}') \end{array}}{(ts,\ m,\ \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow} (ts[i := (p,\ is',\ \vartheta',\ \mathcal{D}',\ \mathcal{O}')],\ m',\ \mathcal{S}')}$$

Fig. 4. Global transitions of virtual machine

4.2 Virtual Machine

The virtual machine is a sequentially consistent machine without store buffers, maintaining additional ghost state to check for the programming discipline. A thread configuration is a tuple $(p,\ is,\ \vartheta,\ \mathcal{D},\ \mathcal{O})$, with the dirty flag $\mathcal{D}$ indicating whether there may be an outstanding volatile write in the store buffer and the set of owned addresses $\mathcal{O}$. The dirty flag $\mathcal{D}$ has to be clean for all volatile reads. The global configuration of the virtual machine $(ts,\ m,\ \mathcal{S})$ maintains a Boolean map of shared addresses $\mathcal{S}$ (indicating write permission). Addresses in the domain of mapping $\mathcal{S}$ are considered shared and $\mathsf{read\text{-}only}\ \mathcal{S} \equiv \{a.\ \mathcal{S}\ a = \lfloor \mathsf{False} \rfloor\}$ is the set of read-only addresses with respect to $\mathcal{S}$. According to the rules in Figure 4, a global transition of the virtual machine $(ts,\ m,\ \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow} (ts',\ m',\ \mathcal{S}')$ is either a program or a memory step.

The transition rules for its memory system are defined in Figure 5, casually grouping the ghost state into a single component $\mathcal{G}$ for succinctness. In addition we introduce the *safety* judgment $\mathcal{O}s,i \vdash (is,\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd$ in Figure 6, where $\mathcal{O}s$ is the list of ownership sets obtained from the thread list ts and i is the index of the current thread. Safety of all reachable states of the virtual machine ensures that the programming discipline is obeyed by the program and is our formal prerequisite for the simulation theorem. It is left as a proof obligation to be discharged by means of a proper program logic for sequentially consistent executions. We elaborate on the rules of Figures 5 and 6 in parallel.

To read from an address it either has to be owned or read-only or it has to be volatile and shared. Moreover a volatile read has to be clean. The memory content of address a is stored in temporary t. A non-volatile write is only allowed if the address is both owned and unshared. The result is written directly into memory. A volatile write is only allowed when no other thread owns the address and the address is not marked as read-only. Simultaneously with the volatile write the dirty flag $\mathcal{D}$ is set and we can transfer ownership as specified by the annotations A, L, R and W. The acquired addresses A must not be owned by any other thread and stem from the shared addresses or are already owned. Reacquiring owned addresses can be used to change the shared-status via the set of local addresses L which have to be a subset of A. The released addresses R have to be owned and distinct from the acquired addresses A. After the write the new ownership set of the thread is obtained by adding the acquired addresses A and releasing the addresses R: $\mathcal{O}' = \mathcal{O} \cup A - R$. The released addresses R are augmented to the shared addresses $\mathcal{S}$ and the local addresses L are removed.

$$\frac{}{(\textsc{Read}\ \mathit{volatile}\ a\ t \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta(t \mapsto m\ a),\ x,\ m,\ \mathcal{G})}$$

$$\frac{}{(\textsc{Write}\ \mathsf{False}\ a\ (D,\ f)\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta,\ x,\ m(a := f\ \vartheta),\ \mathcal{G})}$$

$$\frac{\mathcal{G} = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{True},\ \mathcal{O} \cup A - R,\ \mathcal{S} \oplus_W R \ominus_A L)}{(\textsc{Write}\ \mathsf{True}\ a\ (D,\ f)\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta,\ x,\ m(a := f\ \vartheta),\ \mathcal{G}')}$$

$$\frac{\neg\ \mathit{cond}\ (\vartheta(t \mapsto m\ a)) \qquad \mathcal{G} = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{False},\ \mathcal{O},\ \mathcal{S})}{(\text{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta(t \mapsto m\ a),\ x,\ m,\ \mathcal{G}')}$$

$$\frac{\begin{array}{c}\mathit{cond}\ (\vartheta(t \mapsto m\ a)) \\ \vartheta' = \vartheta(t \mapsto \mathit{ret}\ (m\ a)\ (f\ (\vartheta(t \mapsto m\ a)))) \qquad m' = m(a := f\ (\vartheta(t \mapsto m\ a))) \\ \mathcal{G} = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{False},\ \mathcal{O} \cup A - R,\ \mathcal{S} \oplus_W R \ominus_A L)\end{array}}{(\text{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta',\ x,\ m',\ \mathcal{G}')}$$

$$\frac{\mathcal{G} = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad \mathcal{G}' = (\mathcal{D},\ \mathcal{O} \cup A - R,\ \mathcal{S} \oplus_W R \ominus_A L)}{(\textsc{Ghost}\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}')}$$

$$\frac{\mathcal{G} = (\mathcal{D},\ \mathcal{O},\ \mathcal{S}) \qquad \mathcal{G}' = (\mathsf{False},\ \mathcal{O},\ \mathcal{S})}{(\textsc{Fence} \cdot \mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}) \xrightarrow{\mathsf{v}}_{\mathsf{m}} (\mathit{is},\ \vartheta,\ x,\ m,\ \mathcal{G}')}$$

Fig. 5. Memory transitions of the virtual machine

$$\frac{a \in \mathcal{O} \vee a \in \mathsf{read\text{-}only}\ \mathcal{S} \vee \mathit{volatile} \wedge a \in \mathsf{dom}\ \mathcal{S} \qquad \mathit{volatile} \longrightarrow \neg\ \mathcal{D}}{\mathcal{O}s,i \vdash (\textsc{Read}\ \mathit{volatile}\ a\ t \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{a \in \mathcal{O} \qquad a \notin \mathsf{dom}\ \mathcal{S}}{\mathcal{O}s,i \vdash (\textsc{Write}\ \mathsf{False}\ a\ (D,\ f)\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}\forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \\ a \notin \mathsf{read\text{-}only}\ \mathcal{S} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset \\ A \subseteq \mathcal{O} \cup \mathsf{dom}\ \mathcal{S} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset\end{array}}{\mathcal{O}s,i \vdash (\textsc{Write}\ \mathsf{True}\ a\ (D,\ f)\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\neg\ \mathit{cond}\ (\vartheta(t \mapsto m\ a)) \qquad a \in \mathsf{dom}\ \mathcal{S} \cup \mathcal{O}}{\mathcal{O}s,i \vdash (\text{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}\mathit{cond}\ (\vartheta(t \mapsto m\ a)) \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow a \notin \mathcal{O}s_{[j]} \\ a \notin \mathsf{read\text{-}only}\ \mathcal{S} \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset \\ A \subseteq \mathcal{O} \cup \mathsf{dom}\ \mathcal{S} \qquad L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset\end{array}}{\mathcal{O}s,i \vdash (\text{RMW}\ a\ t\ (D,\ f)\ \mathit{cond}\ \mathit{ret}\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{\begin{array}{c}A \subseteq \mathsf{dom}\ \mathcal{S} \cup \mathcal{O} \\ L \subseteq A \qquad R \subseteq \mathcal{O} \qquad A \cap R = \emptyset \qquad \forall j{<}|\mathcal{O}s|.\ i \neq j \longrightarrow A \cap \mathcal{O}s_{[j]} = \emptyset\end{array}}{\mathcal{O}s,i \vdash (\textsc{Ghost}\ A\ L\ R\ W \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

$$\frac{}{\mathcal{O}s,i \vdash (\textsc{Fence} \cdot \mathit{is},\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd} \qquad \frac{}{\mathcal{O}s,i \vdash ([],\ \vartheta,\ m,\ \mathcal{D},\ \mathcal{O},\ \mathcal{S})\ \surd}$$

Fig. 6. Safe configurations of a virtual machine

We also take care about the write permissions in the shared state: the released addresses in set W as well as the acquired addresses are marked writable: $\mathcal{S}' = \mathcal{S} \oplus_W R \ominus_A L$. The auxiliary ternary operators to augment and subtract addresses from the sharing map are defined as follows:

$$\mathcal{S} \oplus_W R \equiv \lambda a.\ \texttt{if}\ a \in R\ \texttt{then}\ \lfloor a \in W \rfloor\ \texttt{else}\ \mathcal{S}\ a$$
$$\mathcal{S} \ominus_A L \equiv \lambda a.\ \texttt{if}\ a \in L\ \texttt{then}\ \bot\ \texttt{else case}\ \mathcal{S}\ a\ \texttt{of}\ \bot \Rightarrow \bot \mid \lfloor w \rfloor \Rightarrow \lfloor a \in A \vee w \rfloor$$

The read-modify-write instruction has the same operational behavior on the physical state as in the store-buffer machine. As it is an interlocked operation which flushes the store buffer the dirty flag $\mathcal{D}$ is cleaned. The other effects on the ghost state and the safety side-conditions are the same as for the volatile read and volatile write, respectively.

The ghost instruction allows to transfer ownership when no write is involved i.e., when merely reading from memory. It has the same safety requirements as the corresponding parts in the write instructions. The effect of the fence instruction in the virtual machine is to clean the dirty flag. Finally, an empty list of instructions is considered safe.

4.3 Reduction

The reduction theorem reduces a computation of the store buffer machine to a sequential consistent computation of the virtual machine. We formulate this as a simulation theorem which states that a computation of the store buffer machine $(ts_{\mathsf{sb}}, m) \overset{\mathsf{sb}}{\Rightarrow}^* (ts_{\mathsf{sb}}', m')$ can be simulated by a computation of the virtual machine $(ts, m, \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow}^* (ts', m', \mathcal{S}')$. The theorem considers computations that start in an initial configuration where all store buffers are empty and end in a configuration where all store buffers are empty again. A configuration of the store buffer machine is obtained from a virtual configuration by removing all ghost components and assuming empty store buffers. This coupling relation between the thread configurations is written as $ts_{\mathsf{sb}} \sim ts$. Moreover, the precondition $\mathsf{initial}_{\mathsf{v}}\ ts\ \mathcal{S}$ ensures that the ghost state of the initial configuration of the virtual machine is set up properly: the ownership sets of the threads are distinct, an address marked as read-only (according to $\mathcal{S}$) is unowned and every unowned address is shared. Finally, we ensure conformance to the programming discipline by the precondition $\mathsf{safe\text{-}reach}\ (ts, m, S)$ which guarantees that *all reachable* configurations in the virtual machine are safe (according to the rules in Figure 6).

Theorem 1 (Reduction)

$$(ts_{\mathsf{sb}}, m) \overset{\mathsf{sb}}{\Rightarrow}^* (ts_{\mathsf{sb}}', m') \wedge \mathsf{empty\text{-}store\text{-}buffers}\ ts_{\mathsf{sb}}' \wedge ts_{\mathsf{sb}} \sim ts \wedge \mathsf{initial}_{\mathsf{v}}\ ts\ \mathcal{S} \wedge \mathsf{safe\text{-}reach}\ (ts, m, \mathcal{S}) \longrightarrow (\exists\, ts'\ \mathcal{S}'.\ (ts, m, \mathcal{S}) \overset{\mathsf{v}}{\Rightarrow}^* (ts', m', \mathcal{S}') \wedge ts_{\mathsf{sb}}' \sim ts')$$

This theorem captures our intuition that every result that can be obtained from a computation of the store buffer machine can also be obtained by a sequentially consistent computation. However, to prove it we need some generalizations that we describe in a technical report [5]. First of all the theorem is not inductive as we do not consider arbitrary intermediate configurations but only those where all store buffers are empty. For intermediate configurations the coupling relation

becomes more involved. The major obstacle is that a volatile read can overtake non-volatile writes that are still in the store-buffer and have not yet emerged to memory. Keep in mind that our programming discipline only ensures that no *volatile* writes can be in the store buffer the moment we do a volatile read, outstanding non-volatile writes are allowed.

5 Conclusion

We have presented a practical and flexible programming discipline for concurrent programs and have formally proven that it ensures sequential consistency on TSO machines, such as present x64 architectures. Our approach covers a wide variety of concurrency control like locking, data races, single writer multiple readers, read only and thread local portions of memory. We minimize the need for store buffer flushes to optimize the usage of the hardware. Our theorem is not coupled to a specific logical framework like separation logic but is based on more fundamental arguments, namely the adherence to the programming discipline which can be discharged within any program logic using the standard sequentially consistent memory model, without any of the complications of TSO.

Related work. Our model is compatible with the recent revisions of the Intel manuals [9] and the x86 model presented in [12] which revises their previous work [7]. The state of the art in formal verification of concurrent programs is still based on a sequentially consistent memory model. To justify this on a weak memory model a quite drastic approach is chosen, allowing only coarse-grained concurrency usually implemented by locking. Thereby data races are ruled out and data race free programs can be considered as sequentially consistent for example for the Java memory model [14,3] or the x86 memory model [12]. Ridge [13] considers weak memory and data-races and verifies Peterson's mutual exclusion algorithm, ensuring sequential consistency by flushing after every write.

Burckhardt and Musuvathi [4] describe an execution monitor that efficiently checks whether a sequentially consistent TSO execution has a single-step extension that is not sequentially consistent. Like our approach, it avoids having to consider the store buffers as an explicit part of the state. However, their condition requires maintaining enough history information to determine causality between events, which means maintaining a vector clock (which is itself unbounded) for each memory address. Causality (being essentially graph reachability) is already not first-order, and hence unsuitable for many types of program verification.

Closely related to our work is the work of Owens [11] investigating on the conditions for sequential consistent reasoning within TSO. The notion of a *triangular-race* free trace is established to exactly characterize the traces on a TSO machine that are still sequentially consistent. A triangular race occurs between a read and a write of two different threads to the same address, when the reader still has some outstanding writes in the store buffer. To avoid triangular races the reader has to flush the store buffer before reading. This is the same condition that we enforce, if we limit every address to be unowned and every access to be volatile. We regard this limitation as too strong for practical

programs, where non-volatile accesses (without any flushes) to temporarily local portions of memory (e.g. lock protected data) is common practice. This is our core motivation for introducing the ownership based programming discipline.

Limitations. There is a class of important programs that are not sequentially consistent but nevertheless correct. Consider a simple spinlock implementation with a volatile lock `l`, where `l == 0` indicates that the lock is not taken. The following code acquires the lock:

```
while(!interlocked_test_and_set(l));
...critical section accessing protected objects ...,
```

and with the assignment `l = 0` we can release the lock again. Within our framework address `l` can be considered *unowned* (and hence shared) and every access to it is *volatile*. We do not have to transfer ownership of the lock `l` itself but of the objects it protects. As acquiring the lock is an expensive interlocked operation anyway there are no additional restrictions from our framework. The interesting point is the release of the lock via the volatile write `l = 0`. This leaves the dirty bit set, and hence our programming discipline requires a flushing instruction before the next volatile read. If `l` is the only volatile variable this is fine, since the next operation will be a lock acquire again which is interlocked and thus flushes the store buffer. So there is no need for an additional fence. But in general this is not the case and we would have to insert a fence after the lock release to make the dirty bit clean again and to stay sequentially consistent. However, can we live without the fence? For the correctness of the mutual-exclusion algorithm we can, but we leave the domain of sequential consistent reasoning. The intuitive reason for correctness is that the threads waiting for the lock do no harm while waiting. They only take some action if they see the lock being zero again, this is when the lock release has made its way out of the store buffer.

Another example is the following barrier synchronization: each processor has a flag that it exclusively writes (with volatile writes without any flushing) and other processors read, and each processor waits for all processors to set their flags before continuing past the barrier. *Each* processor might first see *its own* flag set and immediately or later still see *all other* flags clear. This is not sequentially consistent, as each processor observes a different order, but it is still correct.

Common for these examples is that there is only a single writer to an address, and the values written are monotonic in a sense that allows the readers to draw the correct conclusion when they observe a certain value. This pattern is named *Publication Idiom* in Owens work [11].

Future work. The first direction of future work is to try to deal with the limitations of sequential consistency described above and try to come up with a more general reduction theorem that can also handle non sequentially consistent portions of code that follow some monotonicity rules. Another direction of future work is to take compiler optimization into account. Our volatile accesses correspond roughly to volatile memory accesses within a C program. An optimizing compiler is free to convert any sequence of non-volatile accesses into a (sequentially semantically equivalent) sequence of accesses. As long as execution

is sequentially consistent, equivalence of these programs (e.g., with respect to final states of executions that end with volatile operations) follows immediately by reduction. However, some compilers are a little more lenient in their optimizations, and allow operations on certain local variables to move across volatile operations. In the context of C (where pointers to stack variables can be passed by pointer), the notion of "locality" is somewhat tricky, and makes essential use of C semantically forbidding address arithmetic across memory objects. Finally we plan to integrate the programming discipline into VCC [6], a verifier for concurrent C programs, which currently merely assumes sequentially consistent memory. VCC already distinguishes between volatile and non-volatile accesses and has a notion of ownership managed in ghost state that has to be mapped to our needs. The dirty flag has to be added to VCC's ghost state.

Acknowledgments. We thank Mark Hillebrand for discussions and feedback on this work and extensive comments on this paper.

References

1. Advanced Micro Devices (AMD), Inc. AMD64 Architecture Programmer's Manual: Vol. 1-3, rev. 3.14 (September 2007)
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
3. Aspinall, D., Sevcík, J.: Formalising Java's data race free guarantee. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 22–37. Springer, Heidelberg (2007)
4. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
5. Cohen, E., Schirmer, N.: A better reduction theorem for store buffers. Technical report (2009), http://arxiv.org/abs/0909.4637v1
6. Cohen, E., et al.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 1–22. Springer, Heidelberg (2009)
7. Sarkar, S., et al.: The semantics of x86 multiprocessor machine code. In: POPL '09, January 2009, pp. 379–391. ACM Press, New York (2009)
8. Intel. Intel 64 architecture memory ordering white paper. SKU 318147-001 (2007)
9. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual: Vol. 1-3b, rev. 29 (March 2009)
10. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
11. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: ECOOP 2010 (to appear, 2010)
12. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs 2009. LNCS. Springer, Heidelberg (2009)
13. Ridge, T.: Operational reasoning for concurrent caml programs and weak memory models. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 278–293. Springer, Heidelberg (2007)
14. Sevcík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)

Equations: A Dependent Pattern-Matching Compiler

Matthieu Sozeau

Harvard University
mattam@eecs.harvard.edu

Abstract. We present a compiler for definitions made by pattern matching on inductive families in the Coq system. It allows to write structured, recursive dependently-typed functions as a set of equations, automatically find their realization in the core type theory and generate proofs to ease reasoning on them. It provides a complete package to define and reason on functions in the proof assistant, substantially reducing the boilerplate code and proofs one usually has to write, also hiding the intricacies related to the use of dependent types and complex recursion schemes.

1 Introduction

In this paper, we present a new tool to define and reason on functions manipulating inductive families in the Coq system. At the core of the system is a compiler for dependent pattern-matching definitions given as a set of *equations* into vanilla Coq terms, inspired by the work of Goguen *et al.* [1] . Our system also incorporates with-clauses (as in Epigram or Agda) that can be used to add a pattern on the left-hand side for further refinement and supports structural and well-founded recursion on inductive families using a purely logical and efficient scheme.

The system provides proofs of the equations that can be used as rewrite rules to reason on calls to the function. It also automatically generates the inductive graph of the function and a proof that the function respects it, giving a useful elimination principle for it.

Equations[1] is implemented as an elaboration into the core Coq type theory, allowing the smallest trusted code base possible and ensuring the correctness of the compilation at each use. The whole system makes heavy use of type classes and the high-level tactic language of Coq for greater genericity and extensibility.

The paper is organized as follows: first, we present an implementation of a dependent pattern-matching compiler (§2) supporting with clauses and efficient recursion on inductive families (§3). Then, we show how we can derive support proofs and in particular a powerful elimination principle directly from the structure of our programs (§4). We finally discuss related work in section 5 and conclude (§6).

[1] Available at http://mattam.org/research/coq/equations.en.html

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 419–434, 2010.

2 Dependent Pattern-Matching Compilation Redux

The idea of writing pattern-matching equations over inductive families goes back to Coquand [2]. He introduced the idea of checking that a set of equations formed an exhaustive *covering* of a signature. From this covering one can build an efficient case tree in the standard way [3].

The interesting addition of dependent pattern-matching over simply-typed pattern-matching is the fact that some constructors need not be considered because the type of the object being matched upon guarantees that it could not have been built with them. Moreover, as each constructor refines the indices of a filtered object and as we are considering equations that can have multiple patterns, refinement may have effects on the values or types of other matched objects. This means that each constructor adds static information to the problem, and this process can be used ad libitum, as exemplified by the definition of diag below:

```
Equations {A n} (v : vector (vector A n) n) : vector A n :=
diag A O Vnil := Vnil ;
diag A (S n) (Vcons (Vcons a n v) n v') := Vcons a (diag (vmap vtail v')).
```

We pattern match on a square matrix of size n by n and compute its diagonal. Only two cases need to be considered: either the matrix is empty and so is its diagonal, or the matrix has n + 1 rows made of vectors of size n + 1 and we can extract the element at the top left of the matrix and build the rest of the diagonal recursively.

Internal vs. external approaches. There exist two main approaches to adding dependent pattern matching to a dependent type theory. One is to bake in the high-level pattern matching construct and make the associated coverage checking and unification procedure part of the core system. This is essentially a shallow approach: one works directly in the metalanguage of the system's implementation and avoids building witnesses for the covering and unification. The disadvantages of the external approach are that it makes the trusted code base larger and limits the extensibility of the system: adding a new pattern matching construct like with-clauses requires to modify the kernel's code. Agda implements pattern-matching this way, and there is a proposal to extend Coq in a similar way [4].

The internal approach takes a different path. In this case we use the type theory itself to explain why a definition is correct, essentially building a witness of the covering in terms of the simpler existing constructs on inductive families. This is the path chosen by [1], and the way Epigram implements pattern-matching. One advantage is that the compiler needs not to be trusted: it elaborates a program that can be checked independently in the core type theory. By taking an elaboration viewpoint, it is also much easier to extend the system with new features that can also be compiled away to the core type theory. Our mantra (after McBride) is that type theory is enough to explain high-level programming constructs.

Our implementation closely follows the scheme from [1], its originality comes mainly from a number of design choices that we will explain in detail. We will not present here the whole formal development of pattern-matching compilation as is done in [1] but we will introduce the main structures necessary to describe our contributions.

The compilation process starts from a signature and a set of clauses given by the user, constructed from the grammar given in figure 1.

$$
\begin{array}{lll}
\texttt{term,type} & t,\ \tau & ::= x \mid \lambda x : \tau, t \mid \Pi x : \tau, \tau' \mid \ldots \\
\texttt{binding} & d & ::= (x\ :\ \tau) \mid (x\ :=\ t\ :\ \tau) \\
\texttt{context} & \Gamma, \Delta & ::= \overrightarrow{d} \\
\texttt{program} & prog & ::= \mathsf{f}\ \Gamma\ :\ \tau\ :=\ \overrightarrow{c} \\
\texttt{user clause} & c & ::= \mathsf{f}\ \overrightarrow{up}\ n \\
\texttt{user pattern} & up & ::= x \mid \mathsf{C}\ \overrightarrow{up} \mid \texttt{?}(t) \\
\texttt{user node} & n & ::= :=\ t \mid :=!\ x \mid \texttt{with}\ t := \{\ \overrightarrow{c}\ \}
\end{array}
$$

Fig. 1. Definitions and user clauses

A program is given as a tuple of a (globally fresh) identifier, a signature and a set of user clauses. The signature is simply a list of bindings and a result type. The purposed type of the function f is then $\Pi\ \Gamma, \tau$. Each user clause comprises a set of patterns that will match the bindings Γ and a right hand side which can either be a simple term (program node), an empty node indicating that the type of variable x is uninhabited or a refinement node adding a pattern to the problem, scrutinizing the value of t.

Notations and terminology. We will use the notation $\overline{\Delta}$ to denote the set of variables bound by an environment Δ, in the order of declarations. An *arity* is a term of the form $\Pi\ \Gamma, s$ where s is a sort. A sort (or kind) can be either `Prop` (categorizing propositions) or `Type` (categorizing computational types, like bool). An arity is hence always a type. We consider inductive families to be defined in a (elided) global context by an arity $\mathsf{I} : \Pi\ \Delta, s$ and constructors $\overrightarrow{\mathsf{I}_i : \Pi\ \Gamma_i, \mathsf{I}\ \overrightarrow{t}}$. Although CIC distinguishes between parameters and indices and our implementation does too, we will not distinguish them in the presentation for the sake of simplicity.

Searching for a covering. The goal of the compiler is to produce a proof that the user clauses form an exhaustive covering of the signature, compiling away nested pattern-matchings to simple case splits. As we have multiple patterns to consider and allow overlapping clauses, there may be more than one way to order the case splits to achieve the same results. We use inaccessible patterns (noted `?(`t`)`) as in Agda to help recover a sense of what needs to be destructed and what is statically known to have a particular value, but overlapping clauses force the compilation to be phrased as a search procedure. As usual, we recover a deterministic semantics using a first-match rule when two clauses overlap.

$$
\begin{array}{lll}
\texttt{context map}\ c & ::= \Delta \vdash \overrightarrow{p} : \Gamma \\
\texttt{pattern} & p & ::= x \mid \mathsf{C}\ \overrightarrow{p} \mid \texttt{?(}t\texttt{)} \\
\texttt{splitting} & spl & ::= \mathsf{Split}(c, x, (spl?)^n) \mid \mathsf{Compute}(c, rhs) \\
\texttt{node} & rhs & ::= \mathsf{Program}(t) \mid \mathsf{Refine}(t, c, \ell, spl) \\
\texttt{label} & \ell & ::= \epsilon \mid \ell.n \quad (n \in \mathbb{N})
\end{array}
$$

Fig. 2. Context mappings and splitting trees

The search for a covering works by gradually refining a *programming problem* $\Delta \vdash \overrightarrow{p} : \Gamma$ and building a splitting tree. A programming problem, or context mapping (fig. 2), is a substitution from Δ to Γ, associating to each variable in Γ a pattern p typable in Δ. We start the search with the problem $\Gamma \vdash \overline{\Gamma} : \Gamma$, i.e. the identity substitution on Γ and the list of user clauses. For f with signature $\Pi\ \Delta, \tau$ we define the f-computation type $\mathsf{f_{comp}}\ \Delta := \tau$ and consider the type of the function to be $\Pi\ \Delta, \mathsf{f_{comp}}\ \Delta$ from now on. We use this computation type during compilation to precisely keep track of recursive calls and to interface with tactics. A splitting can either be:

- A $\mathsf{Split}(\Delta \vdash \overrightarrow{p} : \Gamma, x, (s?)^n)$ node denoting that the variable x is an object of an inductive type with n constructors and that splitting it in context Δ will generate n subgoals which are covered by the optional subcoverings s. When the type of x does not unify with a particular constructor's type the corresponding splitting is empty.
- A $\mathsf{Compute}(\Delta \vdash \overrightarrow{p} : \Gamma, rhs)$ node, where the right hand side can be either:
  - A $\mathsf{Program}(t)$ node denoting a leaf program t of type $\mathsf{f}.\ell_{\mathsf{comp}}\ \overrightarrow{p}$ in Δ.
  - A $\mathsf{Refine}(t, c', \ell, s)$ node corresponding to a with rule introducing a pattern for t with s covering the new problem c'. The label ℓ uniquely identifies the node and will be used to define auxiliary definitions.

$$
\begin{array}{ll}
\textsc{Match}(x, p) & := \Uparrow \{x := p\} \\
\textsc{Match}(\mathsf{C}\ \overrightarrow{p}, \mathsf{C}\ \overrightarrow{q}) & := \textsc{Match}(\overrightarrow{p}, \overrightarrow{q}) \\
\textsc{Match}(\mathsf{C}\ \_, \mathsf{D}\ \_) & := \Downarrow \\
\textsc{Match}(\mathsf{C}\ \overrightarrow{p}, y) & := \Rightarrow \{y\} \\
\textsc{Match}(\texttt{?(}t\texttt{)}, \_) & := \Uparrow \emptyset \\
\\
\textsc{Match}(\epsilon, \epsilon) & := \Uparrow \epsilon \\
\textsc{Match}(p_0; \overrightarrow{p}, q_0; \overrightarrow{q}) & := \textsc{Match}(p_0, q_0) \cup \textsc{Match}(\overrightarrow{p}, \overrightarrow{q})
\end{array}
\qquad
\begin{array}{ll}
\Uparrow s \cup \Uparrow s' & := \Uparrow (s \cup s') \\
\Rightarrow vs \cup \Rightarrow vs' & := \Rightarrow (vs \cup vs') \\
\Downarrow \cup \_ \mid \_ \cup \Downarrow & := \Downarrow \\
\Rightarrow vs \cup \_ & := \Rightarrow vs \\
\_ \cup \Rightarrow vs & := \Rightarrow vs
\end{array}
$$

Fig. 3. Matching patterns

Recursively, we will try to match the user patterns of each clause with the current problem $\Delta \vdash \overrightarrow{p} : \Gamma$. Matching patterns $\overrightarrow{q}$ from the a user clause and patterns $\overrightarrow{p}$ from the current programming problem can either fail ($\Downarrow$), succeed ($\Uparrow\ s$) returning a variable substitution s from $\overrightarrow{q}$ to $\overrightarrow{p}$ or get stuck ($\Rightarrow\ vs$) returning a set of variables from $\overrightarrow{p}$ that needs further splitting to match the user patterns in $\overrightarrow{q}$.

– If the clause does not match a particular problem we try the next one, if there are no clauses left we have a non-exhaustive pattern-matching.
– If the problem is stuck on the clause, we try to recursively find a splitting after refining a stuck variable x, creating subproblems that correspond to an instantiation of the variable with each possible constructor. We build a $\mathsf{Split}(c, x, s)$ node from the resulting set of splittings if all succeed, or try the next stuck variable.
– If the clause matches we get back a substitution from the user clause variables to Δ, so we can typecheck right-hand side terms in environment Δ. We look at the right-hand side and decide:

  • If it is a program user node, we simply typecheck the program and build a $\mathsf{Program}(\mathrm{t})$ node.
  • If it is an empty node ($:=!\ x$), we refine x and check that this produces no subproblems, building a `Split` node.
  • If it is a with node ($\Leftarrow t \Rightarrow \{\overrightarrow{c}\}$), we typecheck t in Δ finding its type τ_Δ. We then strengthen the context Δ for t, giving us the minimal context Δ^t to typecheck t and the remaining context Δ_t. This strengthening is in fact a context mapping $\Delta^t, x_t : \tau_\Delta, \Delta_t \vdash str : \Delta, x_t : \tau_\Delta$. We can now abstract t from the remaining context to get a new context: $\Delta_{\ell.n} \triangleq \Delta^t, x_t : \tau_\Delta, \Delta_t[t/x_t]$. We check that this context is well-typed after the abstraction, which might not be the case. We also define an abstracted type for the subprogram $\mathrm{f}.\ell.\mathrm{n}_{\mathsf{comp}}\ \Delta_{\ell.n} := \mathrm{f}.\ell_{\mathsf{comp}}\ \overrightarrow{p}[t/x_t]$ and search for a covering of the identity substitution of $\Delta_{\ell.n}$ using updated user clauses $\overrightarrow{c}$. The user clauses are actually transformed to match the strengthening: each c_i must be of the form $\overrightarrow{p}_i\ p_i^x$ where $\overrightarrow{p}_i$ matches $\overrightarrow{p}$. The matching gives us a substitution from the variables of $\overrightarrow{p}$, the patterns at the with node, to new user patterns. We can easily make new user clauses matching the strengthened context $\Delta_{\ell.n}$ by associating to each variable of $\Delta_{\ell.n}$ its associated user pattern and using p_i^x for the new pattern. The result of the covering will then be a splitting s for the problem $c' = \mathtt{idsubst}(\Delta_{\ell.n})$ from which we can build a $\mathsf{Refine}(t, c', \ell.n, s)$ node with a fresh n. Compiling this term will give us a term of type $\Pi\Delta_{\ell.n}, \mathrm{f}.\ell.\mathrm{n}_{\mathsf{comp}}\ \overline{\Delta}_{\ell.n}$. We can apply this term to $\overline{\Delta}^t, t, \overline{\Delta}_t$ to recover a term of type $\mathrm{f}.\ell_{\mathsf{comp}}\ \overrightarrow{p}$ in the original Δ context, providing a witness for the initial $\Delta \vdash \overrightarrow{p} : \Gamma$ problem. Consider for example the following definition:

```
Equations {A} (l : list A) (p : A → bool) : list A :=
filter A nil p := nil ;
filter A (cons a l) p with p a := {
filter A (cons a l) p true := a :: filter l p ;
filter A (cons a l) p false := filter l p }.
```

When interpreting the `with` node, the patterns of the inner clauses are transformed to match the variables $A\ a\ l\ p$ bound at the `with` node and their additional pattern for $p\ a$.

The compiled term built from this covering will have a type convertible with:

$$\Pi\, A\ a\ l\ p\ (b : \mathsf{bool}), \mathrm{filter}_{\mathsf{comp}}\ A\ (\mathsf{cons}\ a\ l)\ p$$

We can then instantiate b with $p\ a$ to build a term in the initial A, a, l, p context.

This is a basic overview of the algorithm for type-checking pattern-matching definitions as described in [5], except for the treatment of inaccessible patterns we omitted for brevity. In our case however we not only check that pattern-matchings are well-formed, we also produce witnesses for this compilation in the core language, following [1]. Now that we have compiled the program to a simplified splitting tree, we just need to construct a mapping from splittings to Coq terms. We already explained how $\mathsf{Refine}(c, x, \ell, s)$ nodes are compiled, and $\mathsf{Program}(\mathrm{t})$ nodes are trivially compiled, so we just need to map `Split` nodes.

2.1 A Few Constructions

The dependent pattern-matching notation acts as a high-level interface to a unification procedure on the theory of constructors and uninterpreted functions. Our main building block in the compilation process is hence a mechanism to produce witnesses for the resolution of constraints in this theory, and use these to compile `Split` nodes. The proof terms will be formed by applications of simplification combinators dealing with substitution and proofs of injectivity and discrimination of constructors, their two main properties.

The design of this simplifier is based on the "specialization by unification" method developed in [6,7]. The problem we face is to eliminate an object x of type $\mathsf{I}\,\overrightarrow{t}$ in a goal $\Gamma \vdash \tau$ potentially depending on x. We want the elimination to produce subgoals for the allowed constructors of this family instance. To do that, we generalize the goal by fresh variables Δ $(x' : \mathsf{I}\,\overline{\Delta})$ and a set of equations asserting that x' is equal to x, giving us a new, equivalent goal:

$$\Delta, x' : \mathsf{I}\ \overline{\Delta}, \Gamma \vdash \overrightarrow{\overline{\Delta}_i \simeq \overline{t}_i} \to x \simeq x' \to \tau$$

Note that the equations relate terms that may be in different types due to the fresh indices, hence we use heterogeneous equality $\simeq$ to relate them. We can apply the standard eliminator for I on x' in this goal to get subgoals corresponding to all its constructors, all starting with a set of equations relating the indices t of the original instance to the indices of the constructor. We use a recursive tactic to simplify these equalities, solving the impossible cases automatically. Its completeness is asserted in [1]: at the end of specialization we get refined goals where the initial x has been substituted by the allowed constructors only.

Our tactic relies on a set of combinators for simplifying equations in the theory of constructors, most of which are just rephrasings of the substitution principles for Leibniz and heterogeneous equality. The only interesting bit is a simplifier for equalities between constructors. We need a tactic that can simplify any equality $\mathsf{C}\,\overrightarrow{t} = \mathsf{D}\,\overrightarrow{u}$, either giving us equalities between arguments $\overrightarrow{t}$ and $\overrightarrow{u}$ that can be

further simplified or deriving a contradiction if C is different from D. McBride *et al.* [7] describe a generic method to derive such an eliminator that we adapted to Coq. For any (computational) inductive type $\mathsf{I} : \Pi\ \Gamma, \mathtt{Type}$, we can derive a transformer $\mathrm{NoConfusion}_{\mathsf{I}} : \Pi\ \Gamma\ (P : \mathtt{Type}), \mathsf{I}\ \overline{\Gamma} \to \mathsf{I}\ \overline{\Gamma} \to \mathtt{Type}$ that describes how to simplify the goal P under the assumption that the two instances of $\mathsf{I}\ \overline{\Gamma}$ are equal. E.g., for natural numbers we define:

```
Equations NoConfusion_nat (P : Type) (x y : nat) : Type :=
NoConfusion_nat P O O := P → P ;
NoConfusion_nat P (S n) (S m) := (n = m → P) → P ;
NoConfusion_nat P _ _ := P.
```

Suppose we have a goal P and a proof of NoConfusion_nat $P\ x\ y$ for x and y in constructor form supposing $x = y$. The proof will always unfold to an implication ending in P, so we can apply it to our goal. Depending on the form of x and y, we will make the goal progress in different ways. If x and y are both O, then we are left to prove the same goal unchanged, the equality is trivial ($P \to P$). If x and y are both of the form S _ then we are left with a proof of the goal under the additional hypothesis that the arguments are equal ($(n = m \to P) \to P$). Finally, in all other cases, the goal is directly discharged, as we have a witness of P by contradiction of the equality of n and m.

We define a new type class [8] to register *NoConfusion* proofs for each type. Instances can be automatically derived for any computational inductive family. We can then build a generic tactic to simplify any equality hypothesis on a registed type using this construction, which subsumes the standard `discriminate` and `injection` tactics.

Dealing with K. There is one little twist in our simplifier, due to the fact that Coq does not support the principle of "Uniqueness of Identity Proofs", also referred to as Streicher's K axiom [9], which is necessary to compile dependent pattern-matchings:

```
Axiom UIP_refl : ∀ (U : Type) (x : U) (p : x = x), p = eq_refl
```

This principle allows us to simplify a goal depending on a proof p of $x = x$ by substituting the sole constructor eq_refl for p. As we are outside the kernel, we can easily make use of this axiom to do the simplifications, but this means that some of our definitions will not be able to reduce to their expected normal forms: they are not closed in the empty context anymore. We will tame this problem by providing the defining equations as rewrite rules once a function is accepted, making use of the axiom again to prove these.

It is notorious that using rewriting instead of the raw system reduction during proofs is much more robust and lends itself very well to automation. Hence we only lose the ability to compute with these definitions inside Coq itself, for example as part of reflexive tactics. At least two proposed extensions to Coq allow to derive this principle without any axioms: an extension to make dependent pattern-matching more powerful with respect to indices [4] and the addition of proof-irrelevance. Having them would make Equations only more

useful. Note that extracted terms do not suffer from this fact as propositions like equality are erased.

3 Recursion

We now turn to the treatment of recursive definitions. A notorious problem with the Coq system is that it uses a syntactic check to verify that recursive calls are well-formed. Only structurally recursive functions making recursive calls on a single designated argument are allowed. The syntactic criterion is very restrictive, inherently non-modular and a major source of bugs in the core type checker. Its syntactic nature also precludes the use of some program transformations, for example uses of abstraction might turn a guarded program into an unguarded one. To avoid these pitfalls, we can use the same principle as for pattern-matching and *explain* the recursive structure of our programs using type theory itself.

To do so, we will use an elimination principle on the datatype we want to recurse on, that will give us a way to make recursive calls on any subterm. Instead of a syntactic notion of structural recursion, we will now use a logical one, which is compatible with the rest of the logical transformations happening during compilation.

3.1 The Below Way

Goguen *et al.* [1] give a way to elaborate recursive definitions by building a memoizing structure. For any inductive type $\mathsf{I} : \Pi\Gamma, \mathtt{Type}$, we define a new type $\mathrm{Below}_{\mathsf{I}}$ that captures all the recursive subterms of a given term, applied to an arity. For natural numbers, we define Below_nat as follows:

```
Equations Below_nat (P : nat → Type) (n : nat) : Type :=
Below_nat P O := unit ;
Below_nat P (S n) := (P n × Below_nat P n)%type.
```

The Below_nat definition uses the built-in structural recursion to build a tuple of all the recursive subterms of a number, applied to an arbitrary arity P. We can build this tuple for any $n : \mathsf{nat}$ given a functional *step* that builds a $P\ n$ if we have P for all the strict subterms of n, and hence derive an eliminator:

```
Definition rec_nat (P : nat → Type)
  (step : Π n : nat, Below_nat P n → P n) (n : nat) : P n :=
  step n (below_nat P step n).
```

Now suppose we want to define a function by recursion on $n : \mathsf{nat}$. We can simply apply this recursor to get an additional Below_nat $P\ n$ hypothesis in our context. If we then refine n, this Below_nat $P\ n$ hypothesis will unfold at the same time to a tuple of $P\ n'$ for every recursive subterm n' of n. These hypotheses form the allowed recursive calls of the function.

This construction generalizes to inductive families and the predicate can also be generalized by equalities in a similar fashion as the dependent case construct

to allow recursion on subfamilies of a dependent inductive object. For example, consider defining vlast:

```
Equations vlast {A : Type} {n : nat} (v : vector A (S n)) : A :=
vlast A n v by rec v :=
vlast A ?(O) (Vcons a ?(O) Vnil) := a ;
vlast A ?(S n) (Vcons a ?(S n) v) := vlast v.
```

Here we use recursion using Below_vector. When we encounter a recursion user node by rec v (witnessed as $\mathsf{Rec}(v, s)$ in the splitting tree), we apply the recursor for the type of v, after having properly generalized it. The recursion hypothesis is hence of the form:

```
Below_vector A (λ (n' : nat) (v' : vector A n'),
  Π n (v : vector A (S n)), n' = S n → v' ≃ v → vlast_comp v) n v
```

When we use non-structural recursion, recursive calls are rewritten as applications of a trivial generic projection operator for the function:

```
vlast_comp_proj : ∀ (A : Type) (n : nat) (v : vector A (S n))
  {vcomp : vlast_comp v} → vlast_comp v
```

The last argument of the projection is implicit and will be filled either automatically by a proof search procedure or interactively by the user. When we typecheck a recursive call, the procedure will try to find a satisfying vlast_comp object in the context, simplifying and applying Below_vector hypotheses.

This method handles the structurally recursive definitions satisfactorily, but it is very inefficient. Indeed, if we try to reduce a program built with this recursor using a call-by-value reduction, there might be an exponential blowup as the object we are recursing on, that is the tuple of all possible recursive calls $\mathsf{below}_{\mathsf{I}}$, will have to be computed for each call. This is not so important if we are using a lazy reduction strategy but it is prohibitive if we want to compute with a call-by-value strategy inside Coq, or compute with the extracted program in ML. Extraction removes the logical parts of a term (in Prop), like the manipulations on equality used during specialization by unification, but in this case the Below object is computational and must be kept.

To avoid this problem, we will use another way of witnessing the subterm relation that is entirely logical.

3.2 Generalized Subterm Relations

Our solution is to define the subterm relation on an inductive family and write functions by well-founded recursion on this relation. The solution is also restricted to inductive types in Type. Indeed we cannot define any irreflexive relation on inductives in Prop, as that would contradict the proof-irrelevance principle consistent with the calculus.

Definition 1 (Subterm relation). *Given a computational inductive type* $\mathsf{I} : \Pi\ \Delta, \mathtt{Type}$ *with constructors* $\mathsf{I}_i : \Pi\ \Gamma_i, \mathsf{I}\ \overrightarrow{t}$, *we define the generalized subterm relation as an inductive type* $\mathsf{I}^{sub} : \Pi\ \Delta_l\ \Delta_r, \mathsf{I}\ \overline{\Delta}_l \to \mathsf{I}\ \overline{\Delta}_r \to \mathtt{Prop}$. *For each*

constructor $\mathsf{I}_i : \Pi\ \Gamma_i, \mathsf{I}\ \overrightarrow{t}$ *and for each binding of* Γ_i *of the form* $(x : \Pi\ \Gamma_x, \mathsf{I}\ \overrightarrow{u})$ *we add a constructor to the relation:* $\mathsf{I}_n^{sub} : \Pi\ \Gamma_x\ \Gamma_i, \mathsf{I}^{sub}\ \overrightarrow{u}\ \overrightarrow{t}\ (x\ \overline{\Gamma}_x)\ (\mathsf{I}_i\ \overline{\Gamma}_i)$.

Before going further, we will simplify our development by considering only homogeneous relations. Indeed we can define for any inductive type $\Pi\ \Delta, \mathsf{I}\ \Delta$ (any arity in general) a corresponding closed type by wrapping the indices Δ in a dependent sum and both the indices and the inductive type in another dependent sum.

Definition 2 (Telescope transformation). *For any context* Δ*, we define packing* $\Sigma(\Delta)$ *and unpacking* $\overline{\Sigma}(\Delta, s)$ *by recursion on the context*[2]*:*

$$\Sigma(\epsilon) = \mathsf{unit} \qquad \Sigma(x : \tau, \Delta) = \Sigma x : \tau, \Sigma(\Delta)$$
$$\overline{\Sigma}(\epsilon, s) = \epsilon \qquad \overline{\Sigma}(x : \tau, \Delta, s) = \pi_1\ s, \overline{\Sigma}(\Delta, \pi_2\ s)$$

The heterogeneous subterm relation can hence be uncurried to form an homogeneous relation on $\Sigma\ i : \Sigma(\Delta), \mathsf{I}\ \overline{\Sigma}(\Delta, i)$.

The traditional notion of well-founded relation as found in the Coq standard library is restricted to homogeneous relations and based on the following notion of accessibility:

```
Inductive Acc {A} (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x.
```

An element of `Acc A R x` contains a proof that any preceding element of x by R (if any) is also accessible. As objects of `Acc` are inductive, there has to a finite proof for the accessibility of x, hence all possible chains $\cdots R\ x_{i-1} x_i, R\ x_i\ x$ have to be finite. A relation is said to be well-founded if all elements of its support are accessible for it. This corresponds (classicaly) to the descending chain condition. We make a class to register well founded relations:

```
Class WellFounded {A : Type} (R : relation A) := wellfounded : ∀ a, Acc R a.
```

It is then trivial to derive a fixpoint combinator by recursion on the accessibility proof, given a step function as before:

```
Definition FixWf `{WF : WellFounded A R} (P : A → Type)
  (step : Π x : A, (Π y : A, R y x → P y) → P x) : Π x : A, P x.
```

Obviously, we can prove that the direct subterm relation defined above is well-founded. It follows by a simple induction on the object and inversion on the subterm proof relating the subterms and the original term. We still need to take the transitive closure of this relation to get the complete subterm relation. Again it is easily shown that transitive closure preserves well-foundedness.

Using this recursion scheme produces more efficient programs, as only the needed recursive calls have to be computed along with the corresponding proofs of the subterm relation. Extraction of `FixWf` is actually a general fixpoint.

We can use the same technique as before to use this fixpoint combinator in `Equations` definitions, we just need to deal with the currying when applying

[2] We omit type annotations for the construction of sums and the projections, they can be easily infered.

it to an object in an inductive family. Consider the application of the fixpoint combinator for vlast again, our initial problem was:

$$\forall\ A\ n\ (v : \mathsf{vector}\ A\ (\mathsf{S}\ n)),\ \mathrm{vlast\_comp}\ A\ n\ v$$

To apply our recursion operator over vectors, we must first prepare for a dependent elimination on v packed with its index n. To do so, we simply generalize by an equality between the packed object and a fresh variable of the packed type, giving us an equivalent goal:

$$A : \mathsf{Type}\ v' : \{index : \mathsf{nat}\ \&\ \mathsf{vector}\ A\ index\}$$
$$==========$$
$$\forall\ n\ (v : \mathsf{vector}\ A\ (\mathsf{S}\ n)),\ v' = \mathsf{existT}\ (\mathsf{S}\ n)\ v \rightarrow \mathrm{vlast\_comp}\ A\ n\ v$$

We can now directly use the fixpoint combinator on the subterm relation for packed vectors with v'. This results in a new goal with an additional induction hypothesis expecting a packed vector and a proof that it is smaller than the initial packed v. Using currying, unpacking of existentials and the dependent elimination simplification tactic, we get back a goal refining the initial problem with the same patterns $A\ n\ v$.

The last step is to provide a proof search procedure to automatically build proofs of the subterm relation, filling the witnesses that appear at recursive calls. We can easily do so using a hint database with the constructors of the $|^{sub}$ relation and lemmas on the transitive closure relation that only allow to use the direct subterm relation on the right to guide the proof search by the refined v, emulating the unfolding strategy of Below.

Measures. We need not restrict ourselves to the subterm relation for building well-founded definitions, we can also use any other available well-founded relation at our hands. A common one is provided by the inverse image relation produced by a function on a given relation, often referred to as a measure when the relation is the less-than order on natural numbers. We leave this generalization for future work.

4 Reasoning Support

We now turn to the second part of the Equations package: the derivation of support definitions to help reasoning on the generated implementations.

4.1 Building Equations

The easiest step is constructing the proofs of the equations as propositional equalities.

Definition 3 (Equations statements). *We recurse on the splitting tree, bookkeeping the current label ℓ, initially ϵ, and for each* $\mathsf{Compute}(\Delta \vdash \overrightarrow{p} : \Gamma, rhs)$ *node we inspect the right-hand side and generate a statement:*

– $\mathsf{Program}(t)$*: the equation is simply* $\Pi\ \Delta, \mathrm{f}.\ell\ \overrightarrow{p}\ = t$.

– Refine$(t, \Delta' \vdash \overrightarrow{v}^x, x, \overrightarrow{v}_x : \Delta^x, x : \tau, \Delta_x, \ell', s)$*: We know that the new programming problem is just a reordering of the variables in* Δ *after having inserted a declaration for the refined object and abstracted the remaining* Δ_x *context. The auxiliary definition* $\mathrm{f}.\ell'$ *produces an object refining this context, we can hence generate an indirection equation for the helper function:* $\Pi\ \Delta, \mathrm{f}.\ell\ \overrightarrow{p} = \mathrm{f}.\ell'\ \overrightarrow{v}^x\ t\ \overrightarrow{v}_x$*. We continue the generation of equations, considering the new programming problem and setting the current label to* ℓ'*.*

All of these goals are solvable by simply unfolding the definition of the function and simplifying the goal: the constructor forms in the leaf patterns direct the reduction. If we didn't use any axioms during the definition, then these follow definitionally. When we encounter axioms in these proofs we simply rewrite using their expected computational behavior.

We create a database of rewrite rules named f with the proofs of these equations, to allow simplification by rewriting, abstracting many steps of reasoning and computation in a single rewrite.

4.2 Induction Principle

The next step is to build the inductive graph of the function. Considering $\mathrm{f} : \Pi\ \Gamma,\ \mathrm{f}_{\mathsf{comp}}\ \overline{\Gamma}$, we want to build an inductive relation $\mathrm{f}_{\mathsf{ind}} : \Pi\ \Gamma,\ \mathrm{f}_{\mathsf{comp}}\ \overline{\Gamma} \rightarrow \mathtt{Prop}$ that relates arguments $\overline{\Gamma}$ to results $\mathrm{f}_{\mathsf{comp}}\ \overline{\Gamma}$.

We first define a function that finds the occurrences of recursive calls in a right-hand side term and abstracts them by variables. This is easy to do given that all the recursive calls are labeled by the trivial $\mathrm{f}_{\mathsf{comp\_proj}}$ projection.

Definition 4 (Abstracting recursive calls). *The* AbsRec*(f, t) operator is defined by recursion on the term* t*, under a local context* Δ*, initially empty. The operator builds a context representing the abstracted recursive calls and a new term using these abstracted calls. By case on* t*:*

– $\mathrm{f}_{\mathsf{comp\_proj}}\ \overrightarrow{t}\ p$ *: We recursively compute the abstractions in* $\overrightarrow{t}$ *giving us a new context* Δ' *and terms* $\overrightarrow{t'}$*. We extend* Δ' *with a fresh declaration* $res := \lambda\ \Delta,\ \mathrm{f}\ \overrightarrow{t'} : \Pi\ \Delta,\ \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t'}$ *and the term becomes* $res\ \overline{\Delta}$*.*
– $\lambda x : \tau, b$ *: Let the result of abstracting* b *in an extended context* $\Delta, x : \tau$ *be* (Δ', b')*, we return* $(\Delta', \lambda x : \tau, b')$*.*
– $f\ e$ *: We simply combine the results of abstracting* f *and* e *separately.*
– $\mathtt{let}\ x := t\ \mathtt{in}\ b$ *: We do the abstractions in* t *resulting in* Δ', t' *and recursively call the abstraction on* b *in a context extended with* $(x := t')$*. We simply combine the resulting contexts and terms.*
– *Otherwise we return the empty context and the term unchanged.*

Once we get the recursive calls abstracted, we will need to add induction hypotheses to the context.

Definition 5 (Induction hypotheses generation). *Given a context* Δ *of results produced by the* AbsRec*(f, t) operator, we define the induction hypotheses context by a simple map on* Δ*, denoted* Hyps(Δ)*. For each binding* $res : \Pi\ \Delta,\ \mathrm{f}_{\mathsf{comp}}\ \overrightarrow{t}$ *we build a new binding* $resind : \Pi\ \Delta,\ \mathrm{f}_{\mathsf{ind}}\ \overrightarrow{t}\ (res\ \overline{\Delta})$*.*

We are now ready to build the inductive graph. We will actually be building graphs for both the toplevel definition and each auxiliary definition, resulting in a mutual inductive type $\overrightarrow{\mathrm{f}.\ell_{\mathsf{ind}} : \Pi\ \Delta_{\ell.n},\ \mathrm{f}.\ell_{\mathsf{comp}}\ \overline{\Delta}_{\ell.n} \to \mathtt{Prop}}$.

Definition 6 (Inductive graph). *We compute the constructors of the* $\mathrm{f}.\ell_{\mathsf{ind}}$ *relation by recursion on the splitting tree:*

- $\mathsf{Split}(c, x, s)$ *: Again splitting nodes are basically ignored, we just collect the constructor statements for the splittings* s*, if any.*
- $\mathsf{Rec}(v, s)$ *: Recursion nodes are also ignored when we compute the inductive graph, after all they just produce different ways to build* $\mathrm{f}_{\mathsf{comp}}$ *objects. We just recurse on the splitting* s*.*
- $\mathsf{Compute}(\Delta \vdash \overrightarrow{p} : \Gamma, rhs)$ *: By case on* rhs*:*
  - $\mathsf{Program}(t)$ *: We abstract the recursive calls of the term using the function* AbsRec*(*f*,* t*) which returns a context* ψ *and a new term* t'*. We return the statement*
$$\Pi\ \Delta\ \Psi\ \textsc{Hyps}(\Psi),\ \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ t'$$
  - $\mathsf{Refine}(t, \Delta' \vdash \overrightarrow{v}^x, x, \overrightarrow{v}_x : \Delta^x, x : \tau, \Delta_x, \ell.n, s)$ *: As for the equation, we just have to do an indirection to the inductive graph of the auxiliary function, but we have to take into account the recursive calls of the refined term too. We compute* $\textsc{AbsRec}(f, t) = (\psi, t')$ *and return:*
$$\Pi\ \Delta^x\ \Delta_x\ \Psi\ \textsc{Hyps}(\Psi)\ (res : \mathrm{f}.\ell.\mathrm{n}_{\mathsf{comp}}\ \overline{\Delta}^x\ t'\ \overline{\Delta}_x)$$
$$\mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}\ \overrightarrow{v}^x\ t'\ \overrightarrow{v}_x\ res \to \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ res$$
    We continue with the generation of the $\mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}$ *graph.*

We can now prove that the function (and its helpers) corresponds to this graph by proving the following lemma:

Theorem 1 (Graph lemma). *We prove* $\Pi\ \Delta_\ell,\ \mathrm{f}.\ell_{\mathsf{ind}}\ \overline{\Delta}_\ell\ (\mathrm{f}.\ell\ \overline{\Delta}_\ell)$ *by following the splitting tree.*

- $\mathsf{Rec}(c, s)$ *: We replay recursion nodes, giving us new ways to prove* $\mathrm{f}_{\mathsf{ind}}$ *that we will use to prove the goals corresponding to induction hypotheses.*
- $\mathsf{Split}(c, x, s)$ *: Each split is simply replayed, empty ones solve the goal directly.*
- $\mathsf{Compute}(\Delta \vdash \overrightarrow{p} : \Gamma, rhs)$ *: At computation nodes our goal will necessarily be simplifiable by an equation because we replayed the whole splitting, i.e. it will have the form* $\Pi\ \Delta,\ \mathrm{f}.\ell_{\mathsf{ind}}\ \overrightarrow{p}\ (\mathrm{f}.\ell\ \overrightarrow{p})$*. By case on* rhs*:*
  - $\mathsf{Program}(t)$ *: We rewrite with the equation for this node and apply one of the constructors for the graph. We will optionally get subgoals for induction hypotheses here if* t *had recursive calls in it. They are solved by a proof search, in exactly the same way as the proofs for the recursive calls were found.*
  - $\mathsf{Refine}(t, \Delta' \vdash \overrightarrow{v} : \Gamma', \ell.n, s)$ *: Here we can rewrite with the indirection equation and apply the indirection constructor for the inductive graph, then solve potential induction subgoals. We will be left with a subgoal for* $\mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}$ *in which me must abstract the refined term* t*. We can then recurse on the goal:* $\Pi \Delta_{\mathrm{f}.\ell.\mathrm{n}}, \mathrm{f}.\ell.\mathrm{n}_{\mathsf{ind}}\ \overrightarrow{v}\ (\mathrm{f}.\ell.\mathrm{n}\ \overrightarrow{v})$*.*

□

4.3 Deriving an Eliminator

Once we have the proof of the graph lemma $\Pi\ \Delta,\ \mathsf{f}_{\mathsf{ind}}\ \overline{\Delta}\ (\mathsf{f}\ \overline{\Delta})$, we can specialize the eliminator of $\mathsf{f}_{\mathsf{ind}}$ which is of the form:

$$\begin{array}{l}\Pi\ (P : \Pi\ \Delta,\ \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta} \to \mathtt{Prop})\overrightarrow{(P_{\mathsf{f}.\ell} : \Pi\ \Delta_{\mathsf{f}.\ell},\ \mathsf{f}.\ell_{\mathsf{comp}}\ \overline{\Delta}_{\mathsf{f}.\ell} \to \mathtt{Prop})} \\ (f : \Pi\ \Gamma, \mathsf{f}_{\mathsf{ind}}\ \overrightarrow{t}\ (\mathsf{f}\ \overrightarrow{u}) \to P\ \overrightarrow{t}\ (\mathsf{f}\ \overrightarrow{u}) \to \cdots) \\ \vdots \\ \Pi\ \Delta\ (r : \mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}), \mathsf{f}_{\mathsf{ind}}\ \overline{\Delta}\ r \to P\ \overline{\Delta}\ r\end{array}$$

This eliminator expects not only one proposition depending on the arguments of the initial call to f but also a proposition for each one of the helpers corresponding to refinement nodes. We can easily define the proposition for $\mathsf{f}.\ell.\mathsf{n}$ in terms of the proposition for $\mathsf{f}.\ell$. Each $P_{\mathsf{f}.\ell.\mathsf{n}}$ is defined as:

$$\begin{array}{l}\lambda\ \Delta^x\ (x : \tau)\ \Delta_x\ (r : \mathsf{f}.\ell.\mathsf{n}_{\mathsf{comp}}\ \overline{\Delta}_{\mathsf{f}.\ell.\mathsf{n}}), \Pi\ \Psi\ \textsc{Hyps}(\Psi)\{\mathsf{f}_{\mathsf{ind}} := P\}, \\ \quad \Pi\ H : x = t', P_{\mathsf{f}.\ell}\ \overrightarrow{p}\ (\mathsf{cast}\ r\ x\ H) \quad \text{where } \textsc{AbsRec}(f, t) = \Psi, t'\end{array}$$

We abstract on the context of the refinement node with its distinguished variable x and on a result r for the subprogram. As we know that this subprogram is called with a particular refined value t, we can assert the equality $x = t$ and cast the result with this equality to get back a term of type $\mathsf{f}.\ell_{\mathsf{comp}}\ \overline{\Delta}^x\ \overline{\Delta}_x$: we are simply doing the inverse of the abstraction of t that happened during the typechecking of the refinement node. Of course, if t itself contains recursive calls, we must also abstract by the corresponding P hypotheses and use t' instead.

We want to eliminate not any $\mathsf{f}_{\mathsf{comp}}\ \overline{\Delta}$ object but specifically f calls. Using the graph lemma proof we can trivially specialize the conclusion to $\Pi\ \Delta, P\ \overline{\Delta}\ (\mathsf{f}\ \overline{\Delta})$

We can also remove the unnecessary hypotheses of the form $\mathsf{f}_{\mathsf{ind}}\ \overrightarrow{t}\ (\mathsf{f}\ \overrightarrow{u})$ appearing in the methods, as they are all derivable from the graph lemma proof.

Finally, we can get rid of all the indirection methods as they are of the form:

$$\Pi\ \Delta_{\mathsf{f}.\ell.\mathsf{n}}\ (r : \mathsf{f}.\ell_{\mathsf{comp}}\ \overline{\Delta}_{\mathsf{f}.\ell.\mathsf{n}})\ \Psi\ \textsc{Hyps}(\Psi)\{\mathsf{f}_{\mathsf{ind}} := P\}, P_{\mathsf{f}.\ell.\mathsf{n}}\ \overline{\Delta}^x\ t'\ \overline{\Delta}_x\ r \to P_{\mathsf{f}.\ell}\ \overrightarrow{p}\ r$$

These are readily derivable given the definitions of the $\mathrm{P}.\mathsf{f}.\ell.\mathsf{n}$ above: the equality hypothesis for the refinement is instantiated by a reflexivity proof, making the cast reduce directly. We are left with a tautology.

The cleaned up eliminator can be applied directly to any goal depending on f, possibly after another generalization by equalities if the call has concrete, non-variable arguments. The elimination will give as many goals as there are Program() nodes in the splitting tree (possibly more than the number of actual user nodes due to overlapping patterns). The context will automatically be enriched by equalities witnessing all the refinement information available and of course induction hypotheses will be available for every recursive call appearing in these and the right hand sides. This gives rise to a very powerful tool to write proofs on our programs, and a lightweight one at the same time: all the details of the splitting, refinement and recursion are encapsulated in the eliminator.

5 Related Work

5.1 Dependent Pattern-Matching

The notions of dependent pattern-matching and coverage building were first introduced by Coquand in his seminal article [2] and the initial Alf implementation. It was studied in the context of an extension of the Calculus of Constructions by Cornes [10] who started the work on inversion of dependent inductive types that was later refined and expanded by McBride [11]. Subsequent work with McKinna and Goguen around Epigram [7,12,1] led to the compilation scheme for dependent pattern-matching definitions which is also at the basis of Equations. Using the alternative external approach, Norell developed the Agda 2 language [5], an implementation of Martin-Löf Type Theory that internalizes not only Streicher's axiom K but also the injectivity of inductive types. In a similar spirit, Barras *et al* [4] propose to extend Coq's primitive elimination rule to handle the unification process.

5.2 Recursion in Type Theory

Our treatment of recursion is comparable to the Function tool by Barthe *et al.* [13] which supports well-founded recursion and also generates an inductive graph and a functional induction principle. Our implementation is however more robust as the input program is sufficiently structured to give a complete procedure to generate the graph, and more powerful in its handling of dependent pattern-matching. It does not however remedy the combinatorial explosion due to the use of catch-all clauses in programs as it also uses an expansion strategy to compile pattern-matching.

Another powerful way to handle non-structural recursion in type theory was developed by Bove et Capretta [14]. The technique, based on the ability to first define the inductive domain of a function and delay the termination argument might now be adaptable in our setting.

5.3 Elaborations into Type Theory

The Program [15] extension of Coq which permits elaboration of Coq programs by separating programming and proving lacked the support for reasoning on definitions after the fact. We hope to combine the subset coercions system of Program inside Equations to get the best of both tools.

The Epigram language also incorporates "views" and the application of arbitrary eliminators with `by` in addition to the `with` construct [12] which were not considered here.

6 Conclusion

Future Work. Our setup should allow to extend the language easily with features like first-class patterns or views and support the application of custom tactics during elaboration. We will also need to consider the general case of mutual

(co)-inductive definitions. We also want to evaluate the effectiveness of this approach on a large example, comparing it to other function definition tools like Function or the HOL function package.

We have presented a new tool for defining programs using dependent-pattern matching in the Coq system, automatically generating a supporting theory to ease post-hoc reasoning on them. The system has a safe architecture, living entirely outside the kernel and allowing easy extension thanks to its reliance on the high-level tactic language and type classes constructs. We hope it provides a more robust, accessible and powerful user interface to the calculus.

References

1. Goguen, H., McBride, C., McKinna, J.: Eliminating dependent pattern matching. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 521–540. Springer, Heidelberg (2006)
2. Coquand, T.: Pattern Matching with Dependent Types. In: Proceedings of the Workshop on Logical Frameworks (1992)
3. Augustsson, L.: Compiling Pattern Matching. In: FPCA, pp. 368–381 (1985)
4. Barras, B., Corbineau, P., Grégoire, B., Herbelin, H., Sacchini, J.L.: A New Elimination Rule for the Calculus of Inductive Constructions. In: Berardi, S., Damiani, F., de'Liguoro, U. (eds.) TYPES 2008. LNCS, vol. 5497, pp. 32–48. Springer, Heidelberg (2009)
5. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2007)
6. McBride, C.: Elimination with a Motive. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 197–216. Springer, Heidelberg (2002)
7. McBride, C., Goguen, H., McKinna, J.: A Few Constructions on Constructors. Types for Proofs and Programs, 186–200 (2004)
8. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
9. Streicher, T.: Semantics of Type Theory. Springer, Heidelberg (1991)
10. Cornes, C.: Conception d'un langage de haut niveau de représentation de preuves: Récurrence par filtrage de motifs, Unification en présesence de types inductifs primitifs, Synthése de lemmes d'inversion. PhD thesis, Université Paris 7 (1997)
11. McBride, C.: Inverting Inductively Defined Relations in LEGO. In: Giménez, E., Paulin-Mohring, C. (eds.) TYPES 1996. LNCS, vol. 1512, pp. 236–253. Springer, Heidelberg (1998)
12. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14, 69–111 (2004)
13. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant. Functional and Logic Programming, 114–129 (2006)
14. Bove, A., Capretta, V.: Modelling general recursion in type theory. Mathematical Structures in Computer Science 15, 671–708 (2005)
15. Sozeau, M.: Un environnement pour la programmation avec types dépendants. PhD thesis, Université Paris 11, Orsay, France (2008)

A Mechanically Verified AIG-to-BDD Conversion Algorithm

Sol Swords and Warren A. Hunt, Jr.

Department of Computer Science
University of Texas
1 University Way, M/S C0500
Austin, Texas 78712-0233
{sswords,hunt}@cs.utexas.edu

Abstract. We present the mechanical verification of an algorithm for building a BDD from an AND/INVERTER graph (AIG). The algorithm uses two methods to simplify an input AIG using BDDs of limited size; it repeatedly applies these methods while varying the BDD size limit. One method is similar to dynamic weakening in that it replaces oversized BDDs by a conservative approximation; the other method introduces fresh variables to represent oversized BDDs. This algorithm is written in the executable logic of the ACL2 theorem prover. The primary contribution is the verification of our conversion algorithm. We state its correctness theorem and outline the major steps needed to prove it.

1 Introduction

In this paper we present the mechanical verification of a tool for building a BDD from an AND/INVERTER graph (AIG) representation of a Boolean function. This tool uses two methods to find simplifications of the AIG that may remove unneeded branches while limiting the size of BDDs; one method is similar to dynamic weakening and the other uses variable substitutions to limit the BDD sizes. These methods are applied iteratively with varying BDD size limits until an iteration limit is exhausted or an exact result is achieved. The primary contribution of this paper is the mechanically-checked verification of this algorithm, which has been used in verifying floating-point addition and multiplication hardware at Centaur Technology [5].

After examining related work, we describe our BDD and AIG packages, which are implemented in the executable logic language of the ACL2 theorem prover [7, 6]. We then present the AIG-to-BDD conversion algorithm and the ACL2 theorem of its correctness, with discussion of the major steps necessary to prove the theorem. We believe that our proof is the first mechanically checked proof of such a conversion procedure.

2 Motivation

This work arose from an effort to verify the floating-point addition unit of Centaur Technology's Nano (64-bit) microprocessor design using ACL2-based

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 435–449, 2010.

verification tools. In our verification flow, the hardware design logic is converted to an AIG representation. In order to verify that the hardware implements its specification, we convert the hardware's AIG representation into BDDs using the conversion algorithm presented here. Our algorithm attempts to avoid the AIG-to-BDD conversion for intermediate nodes where the BDD value is irrelevant to the final result. Splitting the analysis of the adder into cases and using the conversion algorithm to avoid building large intermediate BDDs allows the analysis to be completed successfully.

3 Related Work

In general, it is not a robust strategy to attempt verification tasks by building the BDDs of complex Boolean functions; some functions require exponential space to represent as ordered BDDs and others are highly sensitive to variable ordering. However, the symbolic trajectory evaluation (STE) literature has shown that such an approach can be profitable in certain problem domains when strategies such as abstraction of don't-care inputs [10] or splitting the input space into subcases [1] can prove effective. The case-splitting method is particularly helpful in the analysis of floating-point addition [1].

Dynamic weakening [11] is a technique used in STE to avoid blowup when computing BDDs for nodes that are irrelevant to the final result. When using dynamic weakening in STE, BDD values exceeding a certain size are replaced by the unknown value X, which conservatively approximates any symbolic value in the four-valued logic of STE. Our algorithm uses this type of dynamic weakening as one of two approaches to eliminating oversized BDDs; the other approach is less conservative and may reach exact final results at a lower size threshold than dynamic weakening.

Our algorithm is also similar in spirit to cutpoint-based methods of combinational equivalence checking. Originally such methods were based on exhaustive simulation [2], but modern versions use techniques such as localized BDD sweeping [9]. These techniques allow combinational equivalences of complex expressions to be proved while limiting the size of BDDs used in the computations. These techniques also are applicable to a broader set of verification problems than equivalence checking [8], wherein BDD sweeping techniques are often integrated with a SAT solver operating on AIGs. Compared to these cutpoint-based BDD sweeping techniques, our algorithm is lazier; our algorithm only inserts cutpoints where the BDD created exceeds a size threshold, whereas other methods strategically explore several cutpoint frontiers to achieve more simplification of the AIG structure.

4 Definitions

Our goal is to demonstrate the correctness of our AIG-to-BDD conversion. We will describe the interface functions of the BDD and AIG packages, then introduce the algorithm Aig-To-Bdd. We verify the correctness of Aig-To-Bdd with respect to our specification function A2B, which we will also define.

4.1 BDD Package Specification

We have defined a BDD implementation using user-level ACL2 functions [3]. While our BDD data structure definition differs from that of ROBDD packages such as CUDD [12], the same basic operations are possible on both data structures. The proof of correctness of the AIG-to-BDD conversion algorithm is not dependent on the BDD representation. Therefore we will describe the interface of our BDD package without reference to its implementation.

Well-formed BDD objects are recognized by the ACL2 predicate NORMP. The constant Boolean functions are represented by the Boolean values T and NIL. Propositional logic operations on BDDs are named with a Q- prefix, such as Q-NOT, Q-AND, and Q-ITE. The ith BDD variable of the variable order is produced by QV(i). Function MAX-VAR determines the highest numbered BDD variable that is referenced in a BDD; we make use of this later when we wish to introduce a new BDD variable. EVAL-BDD applies the Boolean function represented by a BDD to a list of Boolean values, producing a Boolean result. We have proven the following correctness properties for these operations.

Theorem 1. QV produces a NORMP object:

$$\text{NORMP}(\text{QV}(i)).$$

Theorem 2. Each of the BDD propositional logic operations produces an output satisfying NORMP when its inputs satisfy NORMP. For example,

$$\begin{aligned} \text{NORMP}(x) &\Rightarrow \text{NORMP}(\text{Q-NOT}(x)), \text{ and} \\ (\text{NORMP}(x) \wedge \text{NORMP}(y)) &\Rightarrow \text{NORMP}(\text{Q-AND}(x, y)). \end{aligned}$$

Theorem 3. EVAL-BDD correctly interprets the semantics of the constant functions and the QV constructor:

$$\begin{aligned} \text{EVAL-BDD}(\text{T}, v) &= \text{T}, \\ \text{EVAL-BDD}(\text{NIL}, v) &= \text{NIL}, \text{ and} \\ \text{EVAL-BDD}(\text{QV}(i), v) &= \textbf{if } \text{NTH}(i, v) \textbf{ then } \text{T} \textbf{ else } \text{NIL}. \end{aligned}$$

Here NTH(i, v) gives the ith element of list v.

Theorem 4. EVAL-BDD correctly interprets the semantics of the BDD propositional logic. For example,

$$\begin{aligned} \text{EVAL-BDD}(\text{Q-NOT}(x), v) &= \neg\text{EVAL-BDD}(x, v) \\ \text{EVAL-BDD}(\text{Q-AND}(x, y), v) &= \text{EVAL-BDD}(x, v) \wedge \text{EVAL-BDD}(y, v). \end{aligned}$$

Theorem 5. Only the first MAX-VAR(x) variables are relevant to the evaluation of x:

$$(\text{MAX-VAR}(x) \leq \text{LEN}(v)) \Rightarrow \text{EVAL-BDD}(x, \text{APPEND}(v, v')) = \text{EVAL-BDD}(x, v).$$

Here LEN(v) gives the length of the list v, and APPEND(v, v') gives the list formed by appending the elements of v onto v'.

Theorem 6. MAX-VAR has the following properties:

$$\begin{aligned}\textsc{Max-Var}(\textsc{t}) = \textsc{Max-VAR}(\textsc{nil}) &= 0,\\ \textsc{Max-Var}(\textsc{Qv}(i)) &= i \text{ when } i \text{ is a natural number},\\ \textsc{Max-Var}(\textsc{Q-Not}(x)) &= \textsc{Max-Var}(x), \text{ and}\\ \textsc{Max-Var}(\textsc{Q-And}(x,y)) &\leq \textsc{Max}(\textsc{Max-Var}(x), \textsc{Max-Var}(y)).\end{aligned}$$

Theorem 7. BDDs are canonical. That is, if two BDDs represent the same Boolean function, then they are equal; equivalently, if they are unequal, then there is some variable assignment under which their evaluations are unequal:

$$\begin{aligned}&\textsc{Normp}(x) \wedge \textsc{Normp}(y) \wedge x \neq y\\ &\qquad \Rightarrow \exists v \;.\; \textsc{Eval-Bdd}(x, v) \neq \textsc{Eval-Bdd}(y, v).\end{aligned}$$

Finally, the function $\textsc{Count-Branches}(x, lim)$ counts the number of unique branching nodes in a BDD x unless it is greater than the limit *lim*. Halting the count when it exceeds *lim* is simply a performance optimization. We use this to heuristically determine if it is too expensive to continue computing with a given BDD.

4.2 AIG Package Specification

As with the BDD package, the AIG package is defined with user-level ACL2 functions, and while its implementation differs from other AIG packages, the interface is similar. In our AIG representation, any ACL2 object may be interpreted as an AIG; there is no well-formedness predicate. All other non-cons objects, such as symbols and numbers, are interpreted as variables. The constant Boolean AIGs are represented as T and NIL as with the BDD package. The node constructors AIG-AND and AIG-NOT make new conjunction or negation nodes, respectively. The function AIG-EVAL evaluates the Boolean function represented by an AIG, using a table which maps the AIG variables to Boolean values; $\textsc{Lookup}(key, tab)$ indexes into such a table. We have proven the following correctness properties for these operations.

Theorem 8. AIG-EVAL correctly interprets the semantics of the constant functions and variables:

$$\begin{aligned}\textsc{Aig-Eval}(\textsc{t}, tab) &= \textsc{t}\\ \textsc{Aig-Eval}(\textsc{nil}, tab) &= \textsc{nil}\end{aligned}$$

and if v is a non-Boolean atom,

$$\textsc{Aig-Eval}(v, tab) = \textsc{Lookup}(v, tab).$$

Theorem 9. AIG-EVAL correctly interprets the semantics of AIG-AND and AIG-NOT:

$$\begin{aligned}\textsc{Aig-Eval}(\textsc{Aig-Not}(x), tab) &= \neg\textsc{Aig-Eval}(x, tab)\\ \textsc{Aig-Eval}(\textsc{Aig-And}(x, y), tab) &= \textsc{Aig-Eval}(x, tab) \wedge \textsc{Aig-Eval}(y, tab).\end{aligned}$$

Our goal for the algorithms we will describe is to build a BDD from an AIG given a table mapping AIG variables to BDDs. The specification for such a transformation is A2B. While this is also an executable function in ACL2, it may perform poorly and is used only as a specification. Given input AIG x and table avt ("AIG variable table") mapping AIG variables to BDD values, we define A2B as follows; here we use operators $\neg, \wedge$ as notation for describing AIG nodes, so that $\neg y$ signifies a negation node with subtree y:

$$\text{A2B}(x, avt) = \begin{cases} x & \text{when } x \text{ is a Boolean} \\ \textsc{Lookup}(x, avt) & \text{when } x \text{ is a variable} \\ \text{Q-}\textsc{Not}(\text{A2B}(y, avt)) & \text{when } x \text{ is } \neg y \\ \text{Q-}\textsc{And}(\text{A2B}(a, avt), \\ \qquad \text{A2B}(b, avt)) & \text{when } x \text{ is } a \wedge b. \end{cases} \quad (1)$$

A well-formed AIG variable table is a table of key-value pairs in which each key is a non-Boolean atom and each value is a Normp object. The predicate Avt-Wfp(avt) ("well-formed predicate") checks this property. Finally, the function Table-Max-Var returns the number of the highest BDD variable present in the values of a table avt. Note that

$$\textsc{Max-Var}(\text{A2B}(x, avt)) \leq \textsc{Table-Max-Var}(avt);$$

this is provable by a simple induction on A2B using Theorem 6.

4.3 ACL2(H) Implementation

The ACL2 execution language lacks many features convenient for implementing efficient BDD and AIG structures and operations. ACL2(H) [3] is an extension that rectifies this by adding several capabilities to the underlying execution engine while leaving the ACL2 logic unchanged. The major such capabilities used in the BDD and AIG packages are hash-consing, function memoization, and applicative hash tables. Hash-consing allows BDDs to be created as canonical objects without complicating the user-level code. Similarly, the function memoization facility allows BDD operations to be written as recursive functions without including specific memoization-related code. Applicative hash tables are used extensively in the AIG-to-BDD conversion algorithm.

Because the logic of ACL2(H) is the same as the logic of ACL2, all of our theorems are provable in ACL2, although the execution performance of our algorithms would be unacceptable. Thus we do not need to assume that ACL2(H) is correctly implemented in order to believe that our algorithms are correct; this assumption is, however, necessary if we wish to believe in the results of executing them in ACL2(H).

5 AIG-to-BDD Conversion Algorithm

The idea behind the conversion algorithm is to attempt to determine whether certain AIG nodes can be eliminated as irrelevant by calculating with BDDs

of limited size. Certainly if it can be determined that one of the conjuncts of an AIG AND node reduces to NIL, then the other conjunct is irrelevant. More generally, if conjunct a represents a Boolean function that implies conjunct b, then $a \wedge b$ equals a, so that b is an irrelevant node. Even when b is too expensive to calculate exactly as a BDD (for example, when it is an output of a multiplier) it can sometimes be proven to be irrelevant. Our algorithm tries two methods to find and eliminate irrelevant nodes. One method conservatively approximates BDDs that are larger than the current size threshold, and the other substitutes fresh variables for oversized BDDs. The top-level algorithm applies these two methods according to an order specified by the user.

5.1 Bounding Method

The first method, BOUND-METHOD, uses upper and lower bound (over- and under-approximate) BDDs to approximate the exact result when it cannot be calculated within the current size limit. Here a is a lower bound for b (and b is an upper bound for a) if for all variable assignments v, EVAL-BDD$(a, v) \Rightarrow$ EVAL-BDD(b, v), or equivalently, Q-AND$(a, b) = a$. Each node value is therefore encoded using two BDDs, just as in the onset/offset representation used for the four-valued logic of symbolic trajectory evaluation [4]. In fact, the two representations are equivalent, with the upper bound equivalent to the onset and the lower bound equivalent to the negation of the offset. The bounding method is essentially the same as STE's dynamic weakening technique [11]; however, we describe the technique here in terms of bounds rather than the onset/offset vocabulary of STE.

Every AIG node has an exact BDD value, but this may be too large to compute. Therefore, we associate each node instead with an upper and lower bound BDD, of which each is smaller than a given size threshold. When the upper and lower bounds are identical for a given node, these bounds are equal to the exact BDD of that node. We generate the bounds for a conjunction node by computing Q-AND of the corresponding bounds of its two conjuncts. When the generated upper (resp. lower) bound BDD is larger in size than the current threshold, it is replaced by T (resp. NIL); thus when both the upper and lower bounds exceed the size limit at the same node, that node's value is considered to be the unknown value bounded by NIL and T. If at a conjunction node $a \wedge b$ the upper bound of conjunct a implies the lower bound of conjunct b, then the conjunction equals a. At each size limit, we remove known irrelevant AIG branches; this allows us to avoid reexamining the same irrelevant portions with a larger BDD size limit.

The inputs to BOUND-METHOD consist of the AIG *ain* to be transformed into a BDD, the table *avt* mapping AIG variables to BDDs, the size limit *lim*, and two tables *fmemo* and *bmemo*, used for memoization. The memoization entries are divided between the tables on the basis of whether or not the BDD results are exact; exact results are stored in *fmemo* and inexact ones in *bmemo*. Results from *fmemo* can then be reused, whereas the results in *bmemo* are discarded after each application of this algorithm since they contain approximations specific to a particular BDD size limit. For reference, Fig. 1 lists the tables used in the

Table	Keys	Values
avt	AIG Variables	BDD values
fmemo	Input AIGs	AIG and exact BDD results
smemo	Input AIGs	AIG and BDD results with substitutions
bmemo	Input AIGs	AIG and upper/lower bound BDDs
bvt	Oversized BDDs	Fresh BDD variables

Fig. 1. Tables used in BOUND-METHOD and SUBST-METHOD algorithms

AIG-to-BDD conversion algorithm with their key and value types. The function returns the upper and lower bound BDDs and a new AIG formed by removing irrelevant branches from *ain*. The function proceeds by cases on the syntactic type of *ain*:

Boolean constant. The upper and lower bounds and the AIG result are all equal to *ain*.

Variable. We look up the corresponding BDD value *bv* in *avt*. The size of *bv* is then calculated. If the size is greater than *lim*, the upper and lower bounds returned are T and NIL respectively. Otherwise, the upper and lower bounds are both *bv*. If *bv* is a Boolean constant, then the AIG returned is that constant; otherwise it is *ain*.

Negation. Let x be the negated branch of *ain*. We recursively run the algorithm on x. The upper bound returned is the negation of the lower bound for x, the lower bound is the negation of the upper bound for x, and the new AIG is the negation via AIG-NOT of the AIG returned by the recursive call.

Conjunction. First, we check for an entry for *ain* in the memoization tables. If an entry exists, we return its previously recorded (possibly exact) upper and lower bound BDD and AIG results.

Otherwise, let x and y be the conjoined branches of *ain*. We first run the algorithm recursively on x. Let ux, lx, and ax denote the resulting upper BDD bound, lower BDD bound, and output AIG, respectively. In the case where ux is the constant NIL, we immediately return NIL for all three return values.

Otherwise, we recur also on y. Let uy, ly, and ay denote the resulting upper bound, lower bound, and output AIG, respectively. We check the results to see whether we can determine that *ain* is equal to either x or y. If Q-AND$(ux, ly) = ux$, then $ain = x$ and we return ux, lx, and ax. Similarly if Q-AND$(uy, lx) = uy$, then we return uy, ly, and ay.

If we cannot make such a determination, we compute the upper bound $ub =$ Q-AND(ux, uy) and the lower bound $lb =$ Q-AND(lx, ly). If the size of ub exceeds the limit, it is replaced by T; similarly if the size of lb exceeds the limit it is replaced by NIL. The new AIG returned is the AIG-AND of the return values from the recursive calls unless ub and lb are equal Boolean constants, in which case the AIG returned is that value.

In all of the above cases, we update the memoization tables to reflect the calculated values before returning. If the upper and lower bound BDDs returned are equal, we have an exact result, so *fmemo* is updated to map both *ain* and the result AIG to the resulting AIG and BDD values; otherwise *bmemo* is updated to map *ain* to the result values.

A small example of where this algorithm gives an advantage follows. Consider the AIG $a \wedge \neg(\neg a \wedge b)$. Suppose that the BDD result q_a for a can be calculated exactly. If the upper and lower bound BDDs computed for b are both larger than the size limit, then they are replaced by T and NIL. The bounds for $\neg a$ are calculated exactly as $(\neg q_a, \neg q_a)$. The conjunction of this with b yields bounds $(\neg q_a, \text{NIL})$; the negation yields (T, q_a). Finally, the conjunction of this with a just yields (q_a, q_a) so that the subtree can be replaced by a without calculating a value for b.

A weakness of this method is that it fails to reduce expressions such as $b \wedge \neg b$ when b is too expensive to calculate. The next method is less conservative and therefore can reduce some such expressions.

5.2 Variable Substitution Method

The second, less conservative method replaces each oversized BDD by a fresh BDD variable, associating that oversized BDD with its chosen variable in a table so that if the same BDD is encountered again, the same variable is reused. The variable numbers of fresh variables assigned to oversized BDD results are larger than TABLE-MAX-VAR(*avt*). A BDD result y from this method is an exact value for the target AIG x — that is, $y = \text{A2B}(x, avt)$ — if MAX-DEPTH$(y) \leq$ TABLE-MAX-VAR(*avt*).

The variable-substitution method is implemented by the function SUBST-METHOD. Like BOUND-METHOD, this function takes as inputs the AIG *ain* to be transformed, the table *avt* mapping AIG variables to BDDs, the size limit *lim*, two tables *fmemo* and *smemo* used for memoization, and another table *bvt* ("BDD variable table"). The memoization entries are again divided between the tables based on whether the BDD results are exact. The third table *bvt* maps oversized BDDs to their associated fresh BDD variables. For reference, Fig. 1 lists the tables used in the AIG-to-BDD conversion algorithm with their key and value types. SUBST-METHOD returns a BDD result and an AIG formed by removing irrelevant branches. Again, it proceeds by cases on the syntactic type of *ain*:

Boolean constant. The BDD and AIG result are equal to *ain*.

Variable. We look up the corresponding BDD value in *avt*. We assume this BDD to be smaller than the size bound, and we return it as the BDD result. The AIG result is *ain* itself unless the BDD is a Boolean constant, in which case that value is returned.

Negation. Let x be the negated branch of *ain*. We recursively run the algorithm on x. The BDD returned is the negation of the BDD returned by the recursive call and the AIG returned is the negation via AIG-NOT of the AIG returned by the recursive call.

Conjunction. First, we check for an entry for *ain* in the memoization tables. If an entry exists, we return its previously recorded result BDD and AIG.

Otherwise, let x and y be the conjoined branches of *ain*. We first run the algorithm recursively on x. Let bx and ax be the BDD and AIG result of this recursive call. In the case where bx is NIL, we immediately return NIL for both return values.

Otherwise, we recur also on y. Let by and ay be the BDD and AIG results of this call. We check bx and by to see whether we can determine that *ain* is equal to either x or y. If Q-AND$(bx, by) = bx$ then $ain = x$ and we return bx and ax, and similarly if Q-AND$(bx, by) = by$ then $ain = y$ and we return by and ay.

Otherwise, the conjunction BDD b is Q-AND(bx, by). If the size of b exceeds the limit, we check the *bvt* to see whether b has previously been mapped to a BDD variable. If so, the BDD returned is that variable; otherwise, we assign to b the next fresh variable v, map b to v in *bvt*, and return v. The AIG returned is the conjunction of those returned by the recursive calls unless the conjunction BDD is a constant Boolean, in which case the AIG returned is that value.

In all of the above cases, we update the memoization tables to reflect the calculated values before returning. If the BDD result calculated has MAX-VAR less than or equal to TABLE-MAX-VAR(*avt*), then it is known to be exact and the return values are stored in *fmemo*; otherwise they are stored in *smemo*.

Because this function replaces oversized BDDs by variables rather than by unknowns, it is more accurate than the bounding method. It is capable of resolving some tautologies and contradictions among nodes whose BDD representations are oversized. For example, if B is a subexpression producing an oversized BDD, replacing that BDD with a fresh variable when converting $b \wedge \neg b$ allows that expression to be reduced to NIL, whereas instead bounding it by T and NIL does not enable that reduction. However, SUBST-METHOD is typically slower than BOUND-METHOD at a given size limit.

5.3 Combination of Methods

Our top-level algorithm AIG-TO-BDD, described in Algorithm 1, simply iterates over a user-provided list in which each entry specifies a conversion method and BDD size limit. At each step, we run the specified conversion method at the given size limit. Usually, it is preferable for the size limits to be increasing. If the BDD results from the conversion are determined to be exact, then the loop terminates, returning the AIG and BDD results from the conversion. Otherwise, the simplified AIG resulting from the conversion replaces the input AIG for the next step. The *fmemo* table is retained between iterations since it only collects exact results, whereas the other memoization tables *bmemo* and *smemo* and the BDD variable table *bvt* of the variable substitution method are deleted after each step, since the results stored in these tables depend on the conversion method

Algorithm 1

```
procedure AIG-TO-BDD(ain, avt, steps)
  fmemo ← empty table
  a ← ain
  for step in steps do
    lim ← LIMIT(step)
    if METHOD(step) = BOUND then
      bmemo ← empty table
      (a, ub, lb, fmemo, bmemo) ← BOUND-METHOD(a, avt, lim, fmemo, bmemo)
      if ub = lb then
        return (SUCCESS, ub, a)
      end if
    else                                          ▷ METHOD(step) = SUBST
      smemo ← empty table
      bvt ← empty table
      (a, b, fmemo, smemo, bvt) ← SUBST-METHOD(a, avt, lim, fmemo, smemo, bvt)
      if MAX-VAR(b) ≤ TABLE-MAX-VAR(avt) then
        return (SUCCESS, b, a)
      end if
    end if
  end for
  return (FAILURE, NIL, a)
end procedure
```

and BDD size limit. Finally, if the algorithm reaches the end of the list of steps without obtaining an exact result, then it returns a flag indicating failure along with the simplified AIG produced by the most recent conversion.

6 Correctness Theorem

We prove the following correctness theorem about AIG-TO-BDD:

Theorem 10 (Correctness of Aig-To-Bdd). Let

$$(\mathit{flag}, b, \mathit{aout}) = \text{AIG-TO-BDD}(\mathit{ain}, \mathit{avt}, \mathit{steps}).$$

If AVT-WFP(avt) holds, then:

$$\text{A2B}(\mathit{ain}, \mathit{avt}) = \text{A2B}(\mathit{aout}, \mathit{avt})$$

and, when $\mathit{flag} = \text{SUCCESS}$,

$$\text{A2B}(\mathit{ain}, \mathit{avt}) = b.$$

The proof of this theorem depends on correctness theorems for BOUND-METHOD and SUBST-METHOD, including an important invariant about *fmemo*. We define the predicate FMEMO-CORRECT($\mathit{fmemo}, \mathit{avt}$) to indicate whether all

values stored in *fmemo* are correct, exact results. Specifically, FMEMO-CORRECT checks that for each key *ain* of *fmemo*, the corresponding value is a pair $(aout, b)$ that satisfies

$$\text{A2B}(ain, avt) = \text{A2B}(aout, avt) = b.$$

To prove our top-level theorem by induction, it suffices to show for each method that, when run with the local tables *smemo*, *bmemo*, and *bvt* all empty and *fmemo* satisfying FMEMO-CORRECT,

- the AIG result *aout* is equivalent to *ain* under function composition with *avt*:
$$\text{A2B}(ain, avt) = \text{A2B}(aout, avt),$$
- when an exact BDD result b is found, it is the correct one:
$$\text{A2B}(ain, avt) = b, \text{ and}$$
- the resulting *fmemo* table also satisfies FMEMO-CORRECT.

To prove these properties by induction, however, we state a stronger theorem that combines these three properties with additional invariants about each of the local tables.

For the correctness theorem of BOUND-METHOD, we show that *bmemo* satisfies an invariant BMEMO-CORRECT(*bmemo*, *avt*). This function checks that the inexact upper and lower bounds in each entry of the table are indeed upper and lower bounds of the exact BDD result for the input AIG, and that the reduced AIG result is also equivalent to the input AIG under composition with *avt*. Specifically, BMEMO-CORRECT checks that for each key *ain* in *bmemo*, the corresponding value is a triple $(aout, ub, lb)$ where

$$\text{A2B}(ain, avt) = \text{A2B}(aout, avt)$$

and for all variable assignments v,

$$\begin{aligned} \text{EVAL-BDD}(lb, v) &\Rightarrow \text{EVAL-BDD}(\text{A2B}(ain, avt), v) \text{ and} \\ \text{EVAL-BDD}(\text{A2B}(ain, avt), v) &\Rightarrow \text{EVAL-BDD}(ub, v). \end{aligned}$$

The invariants needed for SUBST-METHOD ensure that *bvt* is well-formed and that the memoized entries stored in *smemo* are correct. The structure of the variable substitution table *bvt* is checked by the function BVT-WFP(*bvt*, *avt*). This table is built up by SUBST-METHOD as new BDD variables are introduced. Each of these new variables must not previously exist in either *avt* or *bvt*, and the BDD to which the variable is associated must only depend on previously introduced variables. Therefore, BVT-WFP checks that the entries in *bvt* may be ordered as

$$(b_{k+1}, \text{QV}(k+1)), (b_{k+2}, \text{QV}(k+2)), \ldots, (b_{k+n}, \text{QV}(k+n))$$

where $k = \textsc{Table-Max-Var}(avt)$ and for each $i \in \{k+1, \ldots, k+n\}$,

$$\textsc{Max-Var}(b_i) < i.$$

This property is preserved each time an oversized BDD is replaced by a variable and the pairing is added to *bvt*, since the new variable is always chosen to be the first BDD variable greater than $\textsc{Table-Max-Var}(avt)$ and not already assigned in *bvt*, and the oversized BDD is constructed from values from *avt* and variables already assigned in *bvt*.

The correctness of the memoization entries stored in *smemo* is checked by the function $\textsc{Smemo-Correct}(smemo, bvt, avt)$. Each entry in *smemo* is a key-value pair similar to those of *fmemo*, where the key is an AIG *ain* and the value is a pairing $(aout, b)$ of a result AIG and a BDD. As with *fmemo* and *bmemo*, Smemo-Correct requires that the result AIG be equivalent to the input AIG under composition with *avt*. However, the BDD result b may depend on variables assigned in *bvt* as well as those in *avt*, due to substitutions of variables for oversize BDDs. Its relation to the exact result $b_{\text{exact}} = \text{A2B}(ain, avt)$ is determined by these substitutions. We define a predicate $\textsc{Bdd-Substp}(b, b_{\text{exact}}, bvt)$ that determines whether b is a representation of b_{exact} under the substitutions given in *bvt*; Smemo-Correct requires that this hold for each entry in *smemo*.

Given an assignment v of values to the variables present in *avt*, we will describe a way to recover the value $\textsc{Eval-Bdd}(b_{\text{exact}}, v)$ from b and *bvt*; Bdd-Substp requires that this algorithm obtains the correct value for all variable assignments. To recover the exact value under an assignment a_k of the first $k = \textsc{Table-Max-Var}(avt)$ variables, we determine an induced value v_i for each substituted variable $\textsc{Qv}(i)$ by evaluating its corresponding oversized BDD b_i, which only references variables of lower indices. That is, we construct extended variable assignments encompassing the substituted variables according to the recurrence

$$\begin{aligned} v_i &= \textsc{Eval-Bdd}(b_i, a_{i-1}) \\ a_i &= \textsc{Append}(a_{i-1}, \textsc{List}(v_i)) \end{aligned} \quad \text{for } i > k.$$

The predicate $\textsc{Bdd-Substp}(b, b_{\text{exact}}, bvt)$ checks whether, for all initial assignments a_k of length k,

$$\textsc{Eval-Bdd}(b, a_n) = \textsc{Eval-Bdd}(b_{\text{exact}}, a_k),$$

where $k+1$ is the smallest index of a substituted variable in *bvt* and n is the number of entries in *bvt*. This is equivalent to requiring that the Boolean function represented by b_{exact} be recoverable from b by resubstituting b_i for $\textsc{Qv}(i)$ for i from $k+n$ down to $k+1$.

Two trivial corollaries of this definition are important for the correctness proof of Subst-Method. If

$$\begin{gathered} \textsc{Bvt-Wfp}(bvt, avt) \text{ and} \\ \textsc{Bdd-Substp}(b, b_{\text{exact}}, bvt), \end{gathered}$$

then

$$\textsc{Max-Var}(b) < k \Rightarrow b = b_{\text{exact}},$$

and for any extension $bvt' \supseteq bvt$,

$$\textsc{Bvt-Wfp}(bvt', avt) \Rightarrow \textsc{Bdd-Substp}(b, b_{\text{exact}}, bvt').$$

We are now ready to state the inductive correctness theorems for Bound-Method and Subst-Method.

Theorem 11 (Inductive Correctness of Bound-Method). Let

$$(aout, ub, lb, fmemo', bmemo') = \textsc{Bound-Method}(ain, avt, lim, fmemo, bmemo)$$

and let

$$b_{\text{exact}} = \text{A2B}(ain, avt).$$

If

$$\textsc{Avt-Wfp}(avt),$$
$$\textsc{Fmemo-Correct}(fmemo, avt), \text{ and}$$
$$\textsc{Bmemo-Correct}(bmemo, avt),$$

then:

$$\textsc{Fmemo-Correct}(fmemo', avt),$$
$$\textsc{Bmemo-Correct}(bmemo', avt),$$
$$\text{A2B}(aout, avt) = b_{\text{exact}},$$
$$\forall v \,.\, (\textsc{Eval-Bdd}(b_{\text{exact}}, v) \Rightarrow \textsc{Eval-Bdd}(ub, v)), \text{ and}$$
$$\forall v \,.\, (\textsc{Eval-Bdd}(lb, v) \Rightarrow \textsc{Eval-Bdd}(b_{\text{exact}}, v)).$$

This theorem shows that Bound-Method preserves the necessary invariants of *fmemo* and *bmemo*, produces BDD results that are upper and lower bounds of the exact BDD, and produces an AIG result that is equivalent to the input under composition with *avt*. Because Bmemo-Correct is trivially satisfied by an empty *bmemo*, this shows that Bound-Method satisfies the properties required for the proof of correctness of Aig-To-Bdd.

Theorem 12 (Inductive Correctness of Subst-Method). Let

$$(aout, b, fmemo', smemo', bvt') = \textsc{Subst-Method}(ain, avt, lim, fmemo, smemo, bvt)$$

and let

$$b_{\text{exact}} = \text{A2B}(ain, avt).$$

If

$$\begin{gathered}\textsc{Avt-Wfp}(avt),\\ \textsc{Fmemo-Correct}(fmemo, avt),\\ \textsc{Bvt-Wfp}(bvt, avt), \text{ and}\\ \textsc{Smemo-Correct}(smemo, bvt, avt),\end{gathered}$$

then:

$$\begin{gathered}bvt \subseteq bvt',\\ \textsc{Bvt-Wfp}(bvt'),\\ \textsc{Smemo-Correct}(smemo', bvt', avt),\\ \textsc{Fmemo-Correct}(fmemo', avt),\\ \text{A2B}(aout, avt) = b_{\text{exact}}, \text{ and}\\ \textsc{Bdd-Substp}(b, b_{\text{exact}}, bvt').\end{gathered}$$

This theorem shows that Subst-Method preserves the necessary invariants of *fmemo*, *bmemo*, and *bvt*, produces a BDD result that is related to the exact result by the substitution given in bvt', and produces an AIG result that is equivalent to the input under composition with *avt*. Because Bvt-Wfp and Smemo-Correct are trivially satisfied when *bvt* and *smemo* are empty, this shows that Subst-Method satisfies the properties required for the proof of correctness of Aig-To-Bdd.

The proof of correctness of the full AIG-to-BDD algorithm involves 150 mechanically checked lemmas in addition to the correctness theorems provided in the BDD and AIG packages. The proof script runs in 22.4 seconds on a 3.00 GHz Intel Xeon E5450 processor.

7 Conclusions

We have presented an algorithm for converting an AIG to a BDD, involving two heuristic simplification methods that are applied while varying a size bound on the BDDs that can be examined. This algorithm is implemented and proven correct in the ACL2 logic, and may be executed efficiently using the ACL2(H) extensions. We have presented the correctness theorem and outlined the major steps of proving that theorem, including the inductive lemmas necessary for proving the correctness of the two subroutines. We believe that our proof is the first tool verification involving this kind of conversion algorithm.

The algorithms described in this paper are used at Centaur Technology to verify several instructions of the VIA Nano processor, including several floating-point addition and multiplication instructions [5]. In many such operations, naive methods of BDD-based symbolic simulation cause blowups because different intermediate signals require different BDD orderings for efficient representation; in these cases, there is no single BDD ordering that can be used to symbolically

simulate the entire circuit. By using our AIG to BDD conversion algorithm and case-splitting in such a way that enough intermediate signals become irrelevant to the final value, we are able to simulate through the circuit with a single BDD ordering per case, allowing us to verify these operations.

Besides giving us confidence in the correctness of our verification tools, having proven the correctness of this algorithm allows us to prove properties of hardware models by symbolic simulation. Our correctness theorem allows us to draw conclusions about the evaluation of an AIG based on the result of converting it to a BDD. This enables a hardware verification methodology in which the result of each completed verification is an ACL2 theorem stating the result, proven without using unverified assumptions about our verification algorithms.

References

1. Aagaard, M.D., Jones, R.B., Seger, C.-J.H.: Formal verification using parametric representations of boolean constraints. In: Proceedings of the 36th Design Automation Conference, pp. 402–407 (1999)
2. Berman, C.L., Trevillyan, L.H.: Functional comparison of logic designs for VLSI circuits. In: IEEE International Conference on Computer-Aided Design, November 5-9, pp. 456–459 (1989)
3. Boyer, R.S., Hunt Jr, W.A.: Function memoization and unique object representation for ACL2 functions. In: ACL '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications, pp. 81–89. ACM, New York (2006)
4. Hazelhurst, S., Seger, C.-J.H.: Symbolic trajectory evaluation. In: Kropf, T. (ed.) Formal Hardware Verification. LNCS, vol. 1287, pp. 3–78. Springer, Heidelberg (1997)
5. Hunt Jr., W.A., Swords, S.: Centaur technology media unit verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 353–367. Springer, Heidelberg (2009)
6. Kaufmann, M., Moore, J.S., Boyer, R.S.: ACL2 version 3.3 (2008), http://www.cs.utexas.edu/~moore/acl2/
7. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell (2000)
8. Kuehlmann, A., Ganai, M.K., Paruthi, V.: Circuit-based boolean reasoning. In: Proceedings of the 38th Design Automation Conference, pp. 232–237 (2001)
9. Kuehlmann, A., Krohm, F.: Equivalence checking using cuts and heaps. In: Proceedings of the 34th Design Automation Conference, June 9-13, pp. 263–268 (1997)
10. Melham, T.F., Jones, R.B.: Abstraction by symbolic indexing transformations. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 1–18. Springer, Heidelberg (2002)
11. Seger, C.-J.H., Jones, R.B., O'Leary, J.W., Melham, T., Aagaard, M.D., Barrett, C., Syme, D.: An industrially effective environment for formal hardware verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24(9), 1381–1405 (2005)
12. Somenzi, F.: CUDD: CU decision diagram package release 2.4.1 (2005), http://vlsi.colorado.edu/~fabio/CUDD/

Inductive Consequences in the Calculus of Constructions*

Daria Walukiewicz-Chrząszcz and Jacek Chrząszcz

Institute of Informatics, Warsaw University
ul. Banacha 2, 02-097 Warsaw, Poland
{daria,chrzaszcz}@mimuw.edu.pl

Abstract. Extending the calculus of constructions with rewriting would greatly improve the efficiency of proof assistants such as Coq. In this paper we address the issue of the logical power of such an extension.

In our previous work we proposed a procedure to check completeness of user-defined rewrite systems. In many cases this procedure demonstrates that only a basic subset of the rules is sufficient for completeness. Now we investigate the question whether the remaining rules are inductive consequences of the basic subset.

We show that the answer is positive for most practical rewrite systems. It is negative for some systems whose critical pair diagrams require rewriting under a lambda. However the positive answer can be recovered when the notion of inductive consequences is modified by allowing a certain form of functional extensionality.

1 Introduction

Theorem provers based on type theory and Curry-Howard Isomorphism, such as Coq [9], are built on top of a powerful computing mechanism which is the lambda calculus. The typing rule which allows for integration of computation into the typing system is called the conversion rule.

Conversion is based on the terminating syntactic reduction relation, it is decidable and hence rather weak. A separate, stronger propositional equality used for reasoning is defined as an inductive predicate with no special treatment in the typing systems. The negative consequences of such approach are the dificulties in handling propositional equalities. For example, if addition is defined by induction on the first argument then even though $\forall x{:}nat,\ x+0=x$ can be proved by induction, the type `vector (n+0)` is not convertible to `vector n` while `vector (0+n)` is.

One solution is to look for new type theories where equality has better reasoning properties without compromising its computational properties [20,2,1]. Another approach is to try to put more power in the computation part, either by adding specific decision procedures to the existing conversion relation, e.g. congruence closure [6], or simply extending the reduction mechanism with user-defined rewriting rules. This paper is concerned with the latter possibility.

* This work was partly supported by Polish government grant N N206 355836.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 450–465, 2010.

Of course adding arbitrary rewrite rules may easily break strong normalization or confluence and hence compromise good meta-theoretical properties of the formalism. Since these properties are undecidable for rewriting systems, there are a number of incomplete decidable criteria that ensure termination and that are flexible enough to be met by many useful rewriting systems [3,5,4,16,17]. Once termination is established an easy test of joinability of critical pairs can tell if the rewriting system is also confluent. A terminating and confluent rewriting system leads to the decidable conversion—it is sufficient to syntactically compare the unique normal forms of inspected terms.

There are two most important roles a rewriting system may play in a theorem prover. First, it can provide a means to decide e.g. a word problem in certain axiomatic theories, for example in group theory. Second, it can be used to define functions, similarly to definitions by pattern matching, but in a more straightforward way. Moreover, a definition by rewriting can contain rules which add more equations to conversion compared to a regular definition by pattern matching. Consider again the addition on unary natural numbers. Using rewriting one can define it in a symmetric way and even include associativity, therefore making the conversion richer and hence the proofs shorter and more automatic.

```
0 + y     ⟶  y
(S x) + y ⟶ S (x + y)
```

```
x + 0     ⟶  x
x + (S y) ⟶ S (x + y)
(x + y) + z ⟶ x + (y + z)
```

In order to trust a proof developed with the help of a proof assistant, one must make sure the development is logically consistent. One can show that it is the case for all developments in which there are no axioms and all definitions by rewriting are complete, i.e. the functions they define are total [4,19]. An automatic procedure to check completeness of definitions by rewriting is also provided in our paper [19].

It turns out that in many cases this procedure uses only a subset of the rules given by the user, demonstrating that this basic subset is already complete. The basic subset roughly corresponds to a definition by cases one can write for example in Coq or a definition using recursors. In this paper we analyse the logical power of the rules that are outside the basic subset to understand how strong can be the generated conversion relation. These additional rules seem to be quite arbitrary even though we know that the whole set of rules is terminating and confluent. Throughout this paper we investigate the question whether the additional rules are inductive consequences of the basic subset.

In the example given above, the set of five rules is strongly normalizing and confluent. The additional three rules (in the right column) are indeed inductive consequences of the basic complete system (in the left column) because all constructor instances of their respective left- and right-hand sides are joinable using the basic set of rules. This can be generalized to all first order rewriting systems (see e.g. Theorem 7.6.5 in [15]).

The standard notion of an inductive consequence [10], meaning an equation which is valid for all ground instances, must of course be adapted to the higher-order setting. Instead of ground instances, we use instances typeable in closed

environments. Our proof technique works by analysing critical pairs formed by additional rules with rules from the basic complete subsystem. In order to transfer the critical pair diagrams to the typed context we impose some slight restrictions on the form of rules and critical pairs.

The first result of the paper concerns the majority of useful rewriting systems: if confluence of critical pairs can be proved without rewriting under a binder, additional rules are inductive consequences of the basic subset (Section 6, Theorem 1). This of course includes all first order rewriting systems, but also definitions over dependently typed symbols and higher-order functions as used in functional programming languages.

The situation is more difficult when rewriting under a binder is necessary to join the critical pair diagrams (Section 7). This is usually the case for functions defined over functional inductive types. To get the positive result in this case (Theorem 2), we must modify the notion of inductive consequences, allowing for a more lax comparison of functional terms, similar in spirit to functional extensionality. Otherwise there are examples where closed instances of left- and right-hand sides of additional rules are not joinable using the basic subset.

From the perspective of a proof assistant user, our results are reassuring about relevance of definitions by rewriting with additional rules. Indeed, for a given definition by rewriting, if the additional rules are shown to be inductive consequences, it means they are valid equations on closed terms, even if they are not necessarily provable as lemmas. Another motivation concerns extraction [12] in a system with rewriting. Although the extraction mechanism can only treat rules which resemble pattern matching, it turns out that the others do not affect the results of computation even for higher order and polymorphic functions.

Because of space considerations, some proofs and examples have been moved from this paper to the web appendix [18].

2 Motivating Examples

In a future version of Coq with rewriting, apart from definitions, axioms and inductive definitions, a user would be allowed to enter definitions by rewriting, even for higher-order and polymorphic functions:[1]

```
Inductive list (A:Set) : Set := nil | cons : A → list A → list A
Symbol map : forall A:Set, (A → A) → list A → list A
Rules
  map A f (nil A) ⟶ nil A
  map A f (cons A a l) ⟶ cons A (f a) (map A f l)
  map A (fun x ⇒ x) l ⟶ l
Parameter l : list nat.
```

The above fragment can be interpreted as an environment consisting of the inductive definition of lists, a definition by rewriting of the `map` function and the

[1] The syntax of the definition by rewriting is inspired by the experimental "recriture" branch of Coq developed by Blanqui. For the sake of clarity we omit certain details, like environments of rule variables.

declaration of a variable `l` of type `list nat`. In this environment all rules for `map` contribute to conversion. They form a terminating and confluent rewriting system in which the first two rules completely define `map`. The third rule is an inductive consequence of the first two (by Theorem 1) and, transformed into equation, can be proved by induction.

Even though we consider higher-order rewriting, we choose the simple matching modulo α-conversion to match a rule to a term. Higher-order matching is useful for example to encode logical languages by higher-order abstract syntax, but it is seldom used in Coq where modeling relies rather on inductive types.

Let us consider another example, the inductive identity function on Brouwer's ordinals:

```
Inductive ord : Set  :=
  o : ord  |  s : ord → ord  |  lim : (nat → ord) → ord.

Symbol id : ord → ord
Rules
  id o ⟶ o
  id (s x) ⟶ s (id x)
  id (lim f) ⟶ lim (fun n ⇒ id (f n))
  id (id x) ⟶ id x
```

This set of rules is also terminating and confluent and the first three rules completely define `id`. The fourth rule says that `id` is an idempotent function. This time, because `ord` is a functional inductive type, the fourth rule cannot be proved to be an inductive consequence of the first three without weakening the notion of inductive consequences. This example will be discussed in details in Section 7. Note however that $\forall x\!:\!\texttt{ord},\ \texttt{id}\ (\texttt{id}\ x) = x$ can be proved in Coq assuming the functional extensionality axiom $\forall f\ g\!:\!A \to B, (\forall x\!:\!A, f\,x = g\,x) \to f = g$.

The last example in this section is the substitutivity property of equality:

```
Inductive  eq (A:Set)(a:A) : A → Prop  :=  refl : eq A a a.

Symbol subst : forall (A: Set)(P: A → Set)(a b: A)(p: eq A a b)(x: P a), P b
Rules
 subst A P a a (refl A a) x ⟶ x
 subst A P a a p x ⟶ x
```

This set of rules is also terminating and confluent and already the first rule completely defines `subst`. The second rule is an inductive consequence of the first one (by Theorem 1) but this time the corresponding equality lemma `subst_eq` $\forall(A : Set)(P : A \to Set)(a : A)(p : eq\,A\,a\,a)(x : P\,a),\ subst\,A\,P\,a\,a\,p\,x = x$, is unprovable in the environment where we have only the standard (like in Coq) elimination of equality. Indeed, one can show[2] that `subst_eq` implies Streicher's axiom K which is not derivable from the standard elimination of equality [11].

[2] See for example http://coq.inria.fr/stdlib/Coq.Logic.EqdepFacts.html

3 Pure Type Systems with Generative Definitions

Even though most papers motivated by the development of Coq concentrate on the calculus of constructions, we present here a slightly more general formalization of a pure type system with inductive definitions and definitions by rewriting. The presentation, taken from [7,8,19], is quite close to the way these elements are and could possibly be implemented in Coq. The formalism is built upon a set of PTS sorts $\mathcal{S}$, a binary relation $\mathcal{A}$ and a ternary relation $\mathcal{R}$ over $\mathcal{S}$ governing the typing rules **(Term/Ax)** and **(Term/Prod)** respectively (Fig. 1). The syntactic class of pseudoterms is defined as follows:

$$t ::= v \mid \mathrm{s} \mid (t_1\ t_2) \mid (\lambda v{:}t_1.t_2) \mid (\forall v{:}t_1.t_2)$$

A pseudoterm can be a variable $v \in \mathit{Var}$, a sort $\mathrm{s} \in \mathcal{S}$, an application, an abstraction or a product. Pseudoterms are identified with finite labelled trees; a λv and a $\forall v$ are binary nodes with the first child corresponding to the type of the variable v and the second to the body of the abstraction (product).

Positions are strings of positive integers. The subterm of t at position p is denoted by $t|_p$ while $t_1[t_2]_p$ stands for the result of replacing $t_1|_p$ with t_2 in t_1. We use $FV(t)$ to denote the set of free variables of a term t. For convenience we assume that all bound variables are different and are different from the free ones.

We use Greek letters γ, δ to denote substitutions which are finite partial maps from variables to pseudoterms. The postfix notation is used for the application of substitutions. We write $[t/x]$ for the substitution of t to a variable x.

Inductive definitions and definitions by rewriting are *generative*, i.e. they are stored in the environment and are used in terms only through names they "generate". An environment is a sequence of declarations, each of them is a variable declaration $v : t$, an inductive definition $\mathsf{Ind}(\Gamma^I := \Gamma^C)$, where Γ^I and Γ^C are environments providing names and types of (possibly mutually defined) inductive types and their constructors, or a definition by rewriting $\mathsf{Rew}(\Gamma, R)$, where Γ is an environment providing names and types of (possibly mutually defined) function symbols and R is a set of rewrite rules defining them. A rewrite rule is a triple denoted by $G \vdash l \longrightarrow r$, where l and r are pseudoterms and G is an environment, assigning types to variables occurring in the left- and right-hand sides l and r. Each l is of the form $f(l_1, \ldots, l_n)$ where $f \in \Gamma$.

Given an environment E, inductive types, constructors and function symbols declared in E are called constants (even though syntactically they are variables). General environments are denoted by E and the environment containing only variable declarations are denoted by Γ, Δ, G, D. We assume that names of all declarations in environments are pairwise disjoint. The set of all variables declared in an environment Γ is denoted by $dom(\Gamma)$.

Given a term t and a position p, we write $\Gamma(t, p)$ to denote the environment of variables that are bound in t on the path from the root to p.

$$\text{Let } \Gamma^I = I_1 : t_1^I \dots I_n : t_n^I \quad \text{and} \quad \Gamma^C = c_1 : t_1^C \dots c_m : t_m^C$$

$$\frac{E \vdash t_j^I : \mathrm{s}_j \quad t_j^I = \forall(\overrightarrow{z : Z_j})\,\mathrm{s}'_j \quad \text{for } j = 1 \dots n \qquad E; \Gamma^I \vdash t_i^C : \hat{\mathrm{s}}_i \quad t_i^C = \forall(\overrightarrow{z : Z'_i})\, I_{j_i}\vec{w} \quad \text{for } i = 1 \dots m}{E \vdash \mathsf{Ind}(\Gamma^I := \Gamma^C) : \mathsf{correct}} \text{ if } \mathsf{POS}_E(\Gamma^I := \Gamma^C)$$

$$\text{Let } \Gamma = f_1 : t_1 \dots f_n : t_n \quad \text{and} \quad R = \{G_i \vdash l_i \longrightarrow r_i\}_{i=1\dots m}, \text{ where } G_i = x_1^i : t_1^i; \dots; x_{n_i}^i : t_{n_i}^i$$

$$\frac{E \vdash t_k : \mathrm{s}_k \quad \text{for } k = 1 \dots n \qquad E; G_i \vdash \mathtt{ok} \quad FV(l_i, r_i) \subseteq G_i \quad \text{for } i = 1 \dots m}{E \vdash \mathsf{Rew}(\Gamma, R) : \mathsf{correct}} \text{ if } \mathsf{ACC}_E(\Gamma, R)$$

$$\frac{}{\epsilon \vdash \mathtt{ok}} \qquad \frac{E \vdash \mathtt{ok} \quad E \vdash t : \mathrm{s}}{E; v : t \vdash \mathtt{ok}}$$

$$\frac{E \vdash \mathtt{ok} \quad E \vdash \mathsf{Ind}(\Gamma^I := \Gamma^C) : \mathsf{correct}}{E; \mathsf{Ind}(\Gamma^I := \Gamma^C) \vdash \mathtt{ok}} \qquad \frac{E \vdash \mathtt{ok} \quad E \vdash \mathsf{Rew}(\Gamma, R) : \mathsf{correct}}{E; \mathsf{Rew}(\Gamma, R) \vdash \mathtt{ok}}$$

$$\frac{E_1; v : t; E_2 \vdash \mathtt{ok}}{E_1; v : t; E_2 \vdash v : t}$$

$$\frac{E \vdash \mathtt{ok}}{E \vdash I_i : t_i^I} \qquad \frac{E \vdash \mathtt{ok}}{E \vdash c_i : t_i^C} \qquad \text{where } \begin{cases} E = E_1; \mathsf{Ind}(\Gamma^I := \Gamma^C); E_2 \\ \Gamma^I = I_1 : t_1^I \dots I_n : t_n^I \\ \Gamma^C = c_1 : t_1^C \dots c_m : t_m^C \end{cases}$$

$$\frac{E \vdash \mathtt{ok}}{E \vdash f_i : t_i} \qquad \frac{E \vdash \mathtt{ok} \quad \delta : G_i \to E}{E \vdash l_i\delta \longrightarrow_R r_i\delta} \qquad \text{where } \begin{cases} E = E_1; \mathsf{Rew}(\Gamma, R); E_2 \\ \Gamma = f_1 : t_1 \dots f_n : t_n \\ R = \{G_i \vdash l_i \longrightarrow r_i\}_{i=1\dots m} \end{cases}$$

(Term/Prod)
$$\frac{E \vdash t_1 : \mathrm{s}_1 \quad E; v : t_1 \vdash t_2 : \mathrm{s}_2}{E \vdash \forall v{:}t_1.t_2 : \mathrm{s}_3} \text{ where } (\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$$

(Term/Abs)
$$\frac{E; v : t_1 \vdash e : t_2 \quad E \vdash \forall v{:}t_1.t_2 : \mathrm{s}}{E \vdash \lambda v{:}t_1.e : \forall v{:}t_1.t_2}$$

(Term/Ax)
$$\frac{E \vdash \mathtt{ok}}{E \vdash \mathrm{s}_1 : \mathrm{s}_2} \text{ where } (\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{A}$$

(Term/App)
$$\frac{E \vdash e : \forall v{:}t_1.t_2 \quad E \vdash e' : t_1}{E \vdash e\ e' : t_2\{v \mapsto e'\}}$$

(Term/Conv)
$$\frac{E \vdash e : t \quad E \vdash t' : \mathrm{s} \quad E \vdash t \approx t'}{E \vdash e : t'}$$

Fig. 1. Definition correctness, environment correctness and lookup, PTS rules

Definition 1. *A* pure type system with generative definitions *is defined by the typing rules in Fig. 1, where:*

- *The relation* $\approx$ *used in the rule* **(Term/Conv)** *is the smallest congruence on well typed terms, generated by* $\longrightarrow$ *which is the sum of beta and rewrite reductions, denoted by* $\longrightarrow_\beta$ *and* $\longrightarrow_R$ *respectively (for the exact definition see [8], Section 2.8).*
- *The notation* $\delta : \Gamma \to E$ *means that* δ *is a* well-typed substitution, *i.e.* $E \vdash v\delta : t\delta$ *for all* $v : t \in \Gamma$.

As in [17,4], recursors and their reduction rules have no special status and they are supposed to be expressed by rewriting.

Assumptions. We assume that we are given a positivity condition POS for inductive definitions and an acceptance condition ACC for definitions by rewriting. Together with the right choice of the PTS they must imply the following properties:

P1 subject reduction, i.e. $E \vdash e : t$, $E \vdash e \longrightarrow e'$ implies $E \vdash e' : t$
P2 uniqueness of types, i.e. $E \vdash e : t$, $E \vdash e : t'$ implies $E \vdash t \approx t'$.
P3 strong normalization, i.e. $E \vdash \mathtt{ok}$ implies that reductions of all well-typed terms in E are finite
P4 confluence, i.e. $E \vdash e : t$, $E \vdash e \longrightarrow^* e'$, $E \vdash e \longrightarrow^* e''$ implies $E \vdash e' \longrightarrow^* \hat{e}$ and $E \vdash e'' \longrightarrow^* \hat{e}$ for some $\hat{e}$.

These properties are usually true in all well-behaved type theories. They are for example all proved for the calculus of algebraic constructions [4], an extension of the calculus of constructions with inductive types and rewriting, where POS is the strict positivity condition as defined in [14], and ACC is the General Schema.

From now on, we use the notation $t\downarrow$ for the unique normal form of t.

4 Completeness of Definitions

The definitions given in this section correspond to the ones given in [19]. Here, for the sake of clarity, we unfold and hence eliminate several auxiliary definitions.

Definition 2 (Canonical form and canonical substitution). *Given a judgment $E \vdash e : t$ we say that the term e is in* canonical *form if and only if:*

- *if $t\downarrow$ is an inductive type then $e = c(e_1, \ldots, e_n)$ for some constructor c and terms $e_1, \ldots, e_n$ in canonical form*
- *otherwise e is arbitrary*

Let Δ be a variable environment and E a correct environment. We call $\delta : \Delta \to E$ canonical *if for every variable $x \in \Delta$, the term $x\delta$ is canonical.*

Definition 3 (Complete definition). *Let E be an environment and $\mathsf{Rew}(\Gamma, R)$ a rewrite definition such that $E \vdash \mathsf{Rew}(\Gamma, R) : \mathsf{correct}$. The definition is* complete*, which is denoted by $\mathsf{COMP}_E(\Gamma, R)$, if and only if for all function symbols $f : (x_1 : t_1) \ldots (x_n : t_n)\, t \in \Gamma$, all environments E' and all canonical substitutions $\delta : (x_1 : t_1; \ldots; x_n : t_n) \to (E; \mathsf{Rew}(\Gamma, R); E')$, such that all terms $x_i\delta$ are in normal form, the term $f(x_1\delta, \ldots x_n\delta)$ is head-reducible by R.*

Below we recall Definition 4.5 and Lemma 4.6 from [19].

Definition 4 (Closed environment). *An environment E is* closed *if and only if it contains only inductive definitions and complete definitions by rewriting, i.e. for each partition of E into $E_1; \mathsf{Rew}(\Gamma, R); E_2$ the condition $\mathsf{COMP}_{E_1}(\Gamma, R)$ is satisfied.*

Lemma 1 (Canonicity). *Let E be a closed environment. If $E \vdash e : t$ and e is in normal form then e is canonical.*

Corollary 1. *Let E be a closed environment such that $E = E_1; \mathsf{Rew}(\Gamma, R); E_2$. Let $f \in \Gamma$ and let $t_1, \ldots, t_n$ be terms such that $E \vdash f(t_1, \ldots, t_n) : t$ for some t. Then $f(t_1, \ldots, t_n)$ is reducible.*

In [19] we also give a sound and terminating, but necessarily incomplete, algorithm that checks whether a rewrite definition is complete. In many cases this algorithm demonstrates that only a basic subset of the rules is sufficient to show completeness.

In the next sections we investigate the question whether the remaining rules are inductive consequences of the basic subset.

5 Towards Inductive Consequences

In equational logic one says that $s = t$ is an inductive consequence of a theory E if for all closed substitutions σ the judgement $E \vdash s\sigma = t\sigma$ holds (see for example [10]).

If equational theories are generated by rewriting systems then the problem of inductive consequences may be reformulated as follows: assuming $R' \supseteq R$, are the rules from $R' - R$ inductive consequences of the rules from R.

In our setting R' is the set of rules given by the user, R is a confluent and complete subsystem of R' and the question is roughly whether for all rules $l \to r \in R' - R$ and all closed substitutions σ, we have $E; \mathsf{Rew}(\Gamma, R) \vdash l\sigma \approx r\sigma$.

In order to show it one must find a sequence of reductions in $E; \mathsf{Rew}(\Gamma, R)$ between $l\sigma$ and $r\sigma$. The proof is done by induction on the sum of the reduction ordering associated with $E; \mathsf{Rew}(\Gamma, R')$ plus the suitable subterm relation. The crucial reduction is the first one from $l\sigma$. It always exists because σ is closed, l starts with a function symbol and $E; \mathsf{Rew}(\Gamma, R)$ is closed. The reduction takes place either entirely in σ or overlaps with l. In the first case one easily gets a new substitution σ' such that $l\sigma$ rewrites to $l\sigma'$ and $r\sigma$ to $r\sigma'$, and one can use the induction hypothesis.

In the second case l and the left-hand side of some rule from $E; \mathsf{Rew}(\Gamma, R)$ overlap and one gets an instance of a critical pair of R'. Since R' is confluent and terminating, its critical pairs are joinable. Following the critical pair diagram one replaces each R' step (which is smaller than $l\sigma$), with a sequence of R steps obtained from the induction hypotheses. In the end one gets the complete R sequence from $l\sigma$ to $r\sigma$.

The usual critical pairs are defined for untyped terms using syntactic unification. The latter has several good properties: it is decidable and it does not introduce new variables.

In order to transfer a critical pair diagram from untyped terms and untyped rewriting to typable terms and rewriting in a PTS, we must slightly restrict the form of rules, which must be left-algebraic (Definition 5), and critical pairs, which must be type compatible (Definition 8). In particular we must be able to compute the types of variables that appear in the left-hand sides of the rules and to check whether these types are compatible (Definition 7) with what is written in the local environments of the rules.

Definition 5 (Algebraic terms, left-algebraic rules). *A term is* algebraic *if every free variable that appears in it is an argument of a constant symbol. A rule $G \vdash l \longrightarrow r$ is* left-algebraic *if l is algebraic.*

Definition 6 (Computed type). *Let t be an algebraic term, p a position and $c : \forall(\overrightarrow{z : Z}).t_c$ a constant such that $t|_p = c(\vec{a})$ for some $\vec{a}$. For a given j the* computed type *for a term $t|_{p \cdot j}$, denoted by $CT(t|_{p \cdot j})$, is $Z_j[a_1/z_1, \ldots a_{j-1}/z_{j-1}]$.*

Definition 7 (Type compatibility). *An environment G is* type compatible *with an algebraic term t if for every $x : T \in G$ there is a position q such that $t|_q = x$ and $T = CT(t|_q)$.*

An environment G is type compatible with a term t if computed types for free variables of t agree with G. It can be understood as a weaker version of typability of t in G.

A well-typed instance of an algebraic, possibly untypable term equipped with a type compatible environment defines a well-typed substitution.

Lemma 2 (Well-typed substitution from well-typed term). *Let t be an algebraic term, G an environment type compatible with t and let ρ be a substitution such that $G' \vdash t\rho : U$ for some U. Then $\rho : G \to G'$ is well-typed.*

For our needs we equip each critical pair with an environment of variables that appear in the pair and we impose the type compatibility assumption on this environment.

Definition 8 (Critical pairs for R' wrt R, type compatibility). *Critical pairs for R' wrt R are critical pairs for every $G \vdash l \longrightarrow r \in R'$ and every $D \vdash g \longrightarrow d \in E; \mathsf{Rew}(\Gamma, R)$ computed using syntactic unification. Critical pairs of $G \vdash l \longrightarrow r$ and $D \vdash g \longrightarrow d$ are tuples $(r\theta, l\theta[d\theta]_p, \Delta)$ for all positions p such that there exists the most general unifier θ unifying $l|_p$ and g and where Δ is the subset of $G; D$, such that $dom(\Delta) = FV(l\theta)$*

A critical pair is type compatible *if Δ is type compatible with $l\theta$.*

Lemma 3 (Unification of left-algebraic rules). *Let $G \vdash l \longrightarrow r$ and $D \vdash g \longrightarrow d$ be left-algebraic rules, p a position in l and θ the most general syntactic unifier of $l|_p$ and g. Then $l\theta$ is algebraic.*

In the next two sections we prove inductive consequence theorems for two kinds of rewriting systems. Section 6 adresses the case of rewriting on non-functional inductive types and Theorem 1 is an extension of results known for first order rewriting. This covers the majority of practical cases, in particular all examples from this paper (and from the web appendix [18]) apart from the definition by rewriting of `id` on `ord`. Section 7 concerns rewriting on functional inductive types.

6 Inductive Consequences

For the rest of this section let us assume that E is a closed environment such that $\mathsf{ACC}_E(\Gamma, R)$ and $\mathsf{COMP}_E(\Gamma, R)$ and that $R' \supseteq R$ extends R with some additional rules in such a way that $\mathsf{ACC}_E(\Gamma, R')$ holds.

At the end of this section Theorem 1 states that in the closed environment $E; \mathsf{Rew}(\Gamma, R)$ the rules from $R' - R$ are inductive consequences of the rules from $E; \mathsf{Rew}(\Gamma, R)$ under the assumption that all applications of rules from $R' - R$ in the critical pair diagrams occur on free positions.

Definition 9 (Free position). *Let t be a term. A position q is free in t if $t|_q$ is not in the scope of any bound variable from t.*

Definition 10 (Free rewriting). *A term s free rewrites to t in the rewriting system $R' - R$, denoted by $E; \mathsf{Rew}(\Gamma, R) \vdash s \xrightarrow{\not\chi R'}_{R'-R} t$, if there exists a rule $G \vdash l \longrightarrow r \in R' - R$ a substitution $\gamma : G \to (E; \mathsf{Rew}(\Gamma, R))$ and a free position q in s such that $s|_q = l\gamma$ and $t = s[r\gamma]_q$.*

Definition 11 ($\not\chi R'$-rewriting). *A term s $\not\chi R'$-rewrites to t, which is denoted by $E; \mathsf{Rew}(\Gamma, R) \vdash s \overset{\not\chi R'}{\rightsquigarrow} t$, if either $E; \mathsf{Rew}(\Gamma, R) \vdash s \longrightarrow t$ or $E; \mathsf{Rew}(\Gamma, R) \vdash s \xrightarrow{\not\chi R'}_{R'-R} t$.*

In other words $\not\chi R'$-rewriting consists in rewriting in the environment $E; \mathsf{Rew}(\Gamma, R)$ with rules from $E; \mathsf{Rew}(\Gamma, R)$ or beta in any context and using rules from $R' - R$ only on free positions.

Definition 12 ($\not\chi R'$-joinability of critical pairs, critical pairs diagram). *We say that (u, v, Δ), a critical pair for R' wrt R, is* $\not\chi R'$-joinable *if it is type compatible and there is a term e such that $\Delta \vdash u \overset{\not\chi R'}{\rightsquigarrow} {}^*e$ and $\Delta \vdash v \overset{\not\chi R'}{\rightsquigarrow} {}^*e$ and if for every $R' - R$ rewrite step $\Delta \vdash s[l'\gamma]_p \longrightarrow s[r'\gamma]_p$ in these sequences the substitution $\gamma : G' \to \Delta$ and the term $l'\gamma$ are well-typed, where G' is the local environment of $G' \vdash l' \longrightarrow r' \in R' - R$.*

The terms u, v, e (with environment Δ) and the aforementioned reductions between them are called a critical pair diagram.

Note that we do not assume that all terms in the critical pair diagrams are typable.

Definition 13 ($\not\chi$-subterm). *Let t be a term. The term $s = t|_q$ is a* $\not\chi$-subterm *of t, denoted by $t \trianglerighteq^{\not\chi} s$, if q is free in t.*

It is well-known that the sum of $\triangleright^{\not\chi}$ and any relation that is well-founded and stable by context is also well-founded.

Theorem 1. *Suppose that critical pairs for R' wrt R are $\not\chi R'$-joinable. Then for every rule $G \vdash l \longrightarrow r \in R' - R$ and substitution $\sigma : G \to (E; \mathsf{Rew}(\Gamma, R))$, such that $E; \mathsf{Rew}(\Gamma, R) \vdash l\sigma : T$ for some T, one has $E; \mathsf{Rew}(\Gamma, R) \vdash l\sigma \approx r\sigma$.*

Proof. By induction on $(\longrightarrow \cup \rhd^{\not\chi})$ where $\longrightarrow$ is the reduction relation corresponding to the environment $E;\mathsf{Rew}(\Gamma,R')$[3]. The relation $\longrightarrow$ is well-founded by assumption $\mathsf{ACC}_E(\Gamma,R')$.

Since $E;\mathsf{Rew}(\Gamma,R)$ is closed and $l\sigma$ is typable, by Corollary 1 the term $l\sigma$ is reducible. There are two possibilities. If the reduction takes place in the substitution then there exists a variable z such that $z:Z\in G$ and $E;\mathsf{Rew}(\Gamma,R)\vdash z\sigma\longrightarrow t'$ for some t'. Let us define σ' to be $\sigma'(x)=\sigma(x)$ for $x\neq z$ and $\sigma'(z)=t'$. The substitution σ' is well-typed since by subject reduction $E;\mathsf{Rew}(\Gamma,R)\vdash t' : Z\sigma'$. The term $l\sigma'$ is also well-typed by subject reduction, since obviously $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\longrightarrow^{+} l\sigma'$. By induction hypothesis applied to $l\sigma'$ we have $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma'\approx r\sigma'$. We have also $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\approx l\sigma'$ and $E;\mathsf{Rew}(\Gamma,R)\vdash r\sigma\approx r\sigma'$ because they result from rewriting with R. By transitivity of $\approx$ we conclude that $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\approx r\sigma$.

Otherwise, there is a rule $D\vdash g\longrightarrow d$ coming from $E;\mathsf{Rew}(\Gamma,R)$ that has a critical pair with $G\vdash l\longrightarrow r$ at position p in l. It means that there exists θ, the most general substitution unifying $l|_p$ and g, and ρ, such that $\sigma=\theta\rho$, and that the critical pair equals $(l\theta[d\theta]_p, r\theta, \Delta)$ where $dom(\Delta)=FV(l\theta)$. Then $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\xrightarrow{\not\chi R'}_{R'-R} r\sigma$ and $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\longrightarrow l\theta[d\theta]_p\rho$. Let us denote $l\theta[d\theta]_p$ by $\hat{l}$. Since critical pairs are joinable there exists a term e such that $E;\mathsf{Rew}(\Gamma,R)\vdash r\sigma\overset{\not\chi R'}{\rightsquigarrow}{}^{*}e\rho$ and $E;\mathsf{Rew}(\Gamma,R)\vdash \hat{l}\rho\overset{\not\chi R'}{\rightsquigarrow}{}^{*}e\rho$.

The term $l\theta$ may be not well-typed but it is algebraic by Lemma 3. We know that the term $l\theta\rho$ is well-typed. By Lemma 2, this implies that ρ is a well-typed substitution from Δ to $E;\mathsf{Rew}(\Gamma,R)$.

Every $R'-R$ step on the path from $r\sigma=r\theta\rho$ or $\hat{l}\rho$ to $e\rho$ is of the form $E;\mathsf{Rew}(\Gamma,R)\vdash s\rho\xrightarrow{\not\chi R'}_{R'-R} t\rho$ where $s=s[l'\gamma]_q$, $t=s[r'\gamma]_q$ for some free position q, a rule $G'\vdash l'\longrightarrow r'\in R'-R$, and a substitution $\gamma: G'\to\Delta$ such that $l'\gamma$ is well-typed in Δ. Of course $s\rho=s\rho[l'\gamma\rho]_q$. Since $\rho:\Delta\to E;\mathsf{Rew}(\Gamma,R)$ is well-typed, the substitution $\gamma\rho$ and the term $l'\gamma\rho$ are also well-typed. Hence we may apply the induction hypothesis to $l'\gamma\rho$ (since it is smaller than $l\sigma$ in $(\longrightarrow\cup\rhd^{\not\chi})^{+}$) and get $E;\mathsf{Rew}(\Gamma,R)\vdash l'\gamma\rho\approx r'\gamma\rho$. Since $\approx$ is stable by context, $E;\mathsf{Rew}(\Gamma,R)\vdash s\rho\approx t\rho$ also holds. Obviously, all rewriting steps corresponding to $E;\mathsf{Rew}(\Gamma,R)$ can be replaced by conversion. Hence $E;\mathsf{Rew}(\Gamma,R)\vdash r\sigma\approx e\rho$ and $E;\mathsf{Rew}(\Gamma,R)\vdash\hat{l}\rho\approx e\rho$. Of course we have also $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\approx\hat{l}\rho$ because $l\sigma$ rewrites to $\hat{l}\rho$ using a rule from R. Consequently by transitivity and symmetry of $\approx$ we get the desired conclusion $E;\mathsf{Rew}(\Gamma,R)\vdash l\sigma\approx r\sigma$.

7 Functional Inductive Consequences

Like in the previous section let us assume that E is a closed environment such that $\mathsf{ACC}_E(\Gamma,R)$ and $\mathsf{COMP}_E(\Gamma,R)$ and that $R'\supseteq R$ extends R with some additional rules in such a way that $\mathsf{ACC}_E(\Gamma,R')$ holds.

[3] Since R-normal canonical forms are not necessarily R'-normal, straightforward R-normalization of $l\sigma$ and $r\sigma$ does not always lead to equal terms.

Like before the goal of this section is to check the power of rules from $R' - R$ with respect to those already present in R. The main difference is that now we allow for systems whose critical pairs need at least one step of rewriting under a binder. This is often the case when considering rewrite rules involving functional inductive types.

Let us consider the `ord` example from Section 2. The rules for `id` are strongly normalizing and confluent. However, to join the critical pair between the 4th and the 3rd rule of `id` one needs rewriting under an abstraction

```
id (id (lim f))  ⟶  id (lim f)  ⟶  lim (fun n ⇒ id (f n))
id (id (lim f))  ⟶  id (lim (fun n ⇒ id (f n)))
  ⟶+  lim (fun n ⇒ id (id (f n)))  ⟶  lim (fun n ⇒ id (f n))
```

It is easy to check that the first three rules for `id` form a complete subsystem. However, it is not true that for every closed substitution σ, term $(\texttt{id (id x)})\sigma$ is convertible with $(\texttt{id x})\sigma$ using only the first three rules. Let `n2o` be defined in the following way:

```
Rewriting n2o : nat -> ord
Rules     n2o 0 ⟶ o          n2o (S x) ⟶ S (n2o x)
```

Consider $\sigma = [\,\texttt{lim (fun n ⇒ n2o n)} \,/\, \texttt{x}\,]$. Then:

```
lσ = id (id (lim (fun n ⇒ n2o n)))  ⟶+  lim (fun n ⇒ id (id (n2o n)))
rσ = id (lim (fun n ⇒ n2o n))  ⟶+  lim (fun n ⇒ id (n2o n))
```

and these are different normal forms. The reason is that even though the substitution is closed, the corresponding instances of the left- and right-hand side of the 4th rule reduce to terms where the function symbol `id` is applied to open terms. In order to pass from open to closed terms again one can consider a new equivalence, containing the usual conversion and identifying functions that are equal for all closed arguments. Let $\sim_\omega$ be the smallest congruence containing $\approx$ and closed by the following (ω) rule:

$$\frac{\begin{array}{c} E \text{ closed} \\ E \vdash f : \forall x{:}A.B \qquad E \vdash g : \forall x{:}A.B \\ \forall d \quad (E \vdash d : A \implies E \Vdash fd \sim_\omega gd) \end{array}}{E \Vdash f \sim_\omega g} \quad (\omega)$$

The rule states roughly that functions f and g are equal if all their closed instances are. It is similar in spirit to functional extensionality in a sense that the $\sim_\omega$ equality is roughly the same to inductive consequences as the propositional equality with functional extensionality to the equality without it.

In fact, we do not need the (ω) rule in its full generality. We will use it only for functions that are arguments of constructors on functional recursive positions, like in `lim (fun n ⇒ n2o n)`. One may also argue that in these places functions are only a means to express infinite branching of a constructor and hence that these functions should be treated extensionally.

Before we can state and prove the theorem corresponding to Theorem 1 from the previous section we need to know more about $\sim_\omega$.

Lemma 4. *Let E be a closed environment and let $s[a]_p$, $s[b]_p$ and T be terms such that $E \vdash s[a]_p : T$, $E \vdash s[b]_p : T$. Moreover, suppose that all declarations in $\Gamma(s,p)$ come from abstractions.*

If $E \Vdash a\delta \sim_\omega b\delta$ holds for all $\delta : \Gamma(s,p) \to E$ then $E \Vdash s[a]_p \sim_\omega s[b]_p$.

The above lemma states that in order to know $E \Vdash s[a]_p \sim_\omega s[b]_p$, which is in some sense an $\sim_\omega$ equality between open terms a and b, it is sufficient to check that $E \Vdash a\delta \sim_\omega b\delta$ holds for all closed substitutions δ. Hence, it shows how to pass from open terms to the closed ones, and in particular from an open instance of the left-hand side of a rule $l'\gamma$ to a closed one $l'\gamma\delta$. Closed instances are necessary since we want to follow the proof of Theorem 1 and use the inductive hypothesis. On the other hand this forces us to use an induction ordering $>$ strong enough to show that $l\sigma > l'\gamma\delta$ for an arbitrary δ, instead of the usual $l\sigma > l'\gamma$.

An example of a well-founded ordering allowing for applications to arbitrary arguments is the constructor subterm ordering on functional types. Taking `ord` for example, the term `lim f` is greater than `f t` for any `t` of type `nat`. Of course this can be done only for recursive arguments of a constructor and because we restrict ourselves to well-typed terms.

Unfortunately constructor subterm is not enough for our needs: we need to use it together with the rewrite relation generated by the environment and with beta reduction. And it is not always the case that the sum of the constructor subterm with a well-founded relation is always well-founded (see an example in [18]). Fortunately, the sum of the constructor subterm with the rewrite relation generated by rules accepted by HORPO is always well-founded [17]. Our hypothesis is that this can be extended to any well-founded relation containing rules for recursors.

Note that $s|_p\delta$ is smaller than s in the constructor subterm ordering only if on the path from the root to q in s there are only constructors and abstractions and that they appear only on recursive positions. For that reason we restrict critical pair diagrams to be joinable that way.

Let us now introduce formally the notions of a recursive position, constructor rewriting and constructor subterm ordering.

Definition 14 (Recursive position). *The i-th position of a constructor $c : \forall(\overrightarrow{p : P})(\overrightarrow{z : d}), I(\vec{p})\vec{w}$ of an inductive type I is* recursive *if d_i is of the form $\forall(x_1 : T_1)\dots(x_n : T_n).I(\vec{p})\vec{v}$. It is called a* nonfunctional recursive position *if $n = 0$; otherwise is is called a* functional recursive position.

Definition 15 (Constructor rewriting). *A term s* constructor rewrites *to t in a rewriting system $R' - R$, which is denoted by $E; \mathsf{Rew}(\Gamma, R) \vdash s \xrightarrow{c}_{R'-R} t$, if there exists a rule $G \vdash l \longrightarrow r \in R' - R$, a position $q = q_1 \cdot \ldots \cdot q_m$, and a substitution $\gamma : G \to (E; \mathsf{Rew}(\Gamma, R))$ such that $s = s[l\gamma]_q$, $t = s[r\gamma]_q$, and for every $k = 0..m-1$*

- *either $s|_{q_1 \dots q_k} = c(\vec{a}, \vec{b})$ for some constructor c and q_{k+1} is a recursive position of c,*

– *or* $s|_{q_1 \dots q_k} = \lambda x : T.s|_{q_1 \dots q_{k+1}}$, $0 < k < m$, $s|_{q_1 \dots q_{k-1}} = c(\vec{a}, \vec{b})$ *for some constructor* c *and* q_k *is a functional recursive position of* c.

Definition 16 (cR'-rewriting). *A term* s cR'*-rewrites to* t*, which is denoted by* $E; \mathsf{Rew}(\Gamma, R) \vdash s \overset{cR'}{\rightsquigarrow} t$*, if either* $E; \mathsf{Rew}(\Gamma, R) \vdash s \longrightarrow t$ *or* $E; \mathsf{Rew}(\Gamma, R) \vdash s \xrightarrow[R'-R]{c} t$.

In other words cR'-rewriting consists in rewriting in the environment $E; \mathsf{Rew}(\Gamma, R)$ with rules from $E; \mathsf{Rew}(\Gamma, R)$ or beta in any context and using $R' - R$ rules only in contexts built from constructors and abstractions as described in Definition 15.

Definition 17 (cR'-joinability of critical pairs, critical pairs diagram). *We say that* (u, v, Δ)*, a critical pair for* R' *wrt* R*, is* cR'-joinable *if it is type compatible and there is a term* e *such that* $\Delta \vdash u \overset{cR'}{\rightsquigarrow} {}^* e$ *and* $\Delta \vdash v \overset{cR'}{\rightsquigarrow} {}^* e$ *and if for every* $R' - R$ *rewrite step in these sequences* $\Delta \vdash s[l'\gamma]_p \longrightarrow s[r'\gamma]_p$

– γ *is a well-typed substitution from* G' *to* $\Delta, \Gamma(s, p)$ *and*
– $l'\gamma$ *is a well-typed term in* $\Delta, \Gamma(s, p)$

where G' is the local environment from $G' \vdash l' \longrightarrow r' \in R' - R$.

The terms u, v, e *(with environment* Δ*) and the aforementioned reductions between them are called a* critical pair diagram.

Since the constructor subterm ordering is not well-founded on nontypable terms, the definition below depends on environment.

Definition 18 (Constructor subterm). *Let* $c : \forall \overrightarrow{(p : P)} \overrightarrow{(z : d)}.I(\vec{p})\vec{w}$ *be a constructor of an inductive type* I *and let* i *be a recursive position of* c*. Let* E *be an environment and* $\vec{a}$, $\vec{b}$ *be terms such that* $c(\vec{a}, \vec{b})$ *is typable in* E.

Then for every $\vec{t}$ *such that* $E \vdash b_i\vec{t} : T$ *for some* T *the term* $b_i\vec{t}$ *is a* constructor subterm of $c(\vec{a}, \vec{b})$ in E*, denoted by* $E \vdash c(\vec{a}, \vec{b}) \rhd^c b_i\vec{t}$.

Theorem 2. *Suppose that all critical pairs for* R' *wrt* R *are* cR'*-joinable and that the relation* $(\longrightarrow \cup \rhd^c)$ *is well-founded in* $E; \mathsf{Rew}(\Gamma, R')$*. Then for every rule* $G \vdash l \longrightarrow r \in R' - R$ *and substitution* $\sigma : G \to (E; \mathsf{Rew}(\Gamma, R))$*, such that* $E \vdash l\sigma : T$ *for some* T*, one has* $E; \mathsf{Rew}(\Gamma, R) \Vdash l\sigma \sim_\omega r\sigma$.

Proof (sketch). By induction on $(\longrightarrow \cup \rhd^c)$ in the environment $E; \mathsf{Rew}(\Gamma, R')$.

The proof follows exactly the schema of the proof of Theorem 1. The difference is that in a critical pair diagram there may be an $R' - R$ rewriting step under a binder, which means that we have an open instance of some left hand-side from $R' - R$ and induction hypothesis cannot be directly applied. We use Lemma 4 to get a closed instance, and then we show that the resulting term is always smaller than $l\sigma$ in the ordering used for induction.

Once we show that induction hypothesis can be applied, the rest of the proof goes as in Theorem 1.

8 Conclusions

In this paper we study the calculus of constructions with rewriting and we address the issue of the logical power of such an extension.

We continue the research about rewriting in the calculus of constructions presented in [19] where an algorithm that checks completeness of definitions by rewriting was given. In many cases this algorithm demonstrates that only a basic subset of the rules is sufficient for completeness. In this paper we have shown that the remaining rules are inductive consequences of the basic subset.

The proof is done for two kinds of rewriting systems: when there is no rewriting under a binder in the critical pairs diagrams (Section 6) and for some class of systems where such situation happens (Section 7). In the latter case the conclusion of the inductive consequences lemma must be modified by allowing for a kind of functional extensionality in the corresponding equivalence.

The additional assumptions on rewriting that we impose do not seem restrictive. First of all we require the rewrite rules to be left-algebraic and the critical pairs to be type-compatible, which can be checked easily. Second, there are assumptions on the form of critical pairs diagrams. These are different in Section 6 and Section 7, where the restrictions are mainly due to difficulties in finding a suitable ordering for induction, but they always account for a simple inspection of a diagram.

It is interesting to relate our paper to the PhD work of Oury [13]. He studies CCE, the extensional calculus of constructions, and shows that CCE is conservative with respect to the calculus of inductive constructions extended with three axioms (functional extensionality, Streicher's axiom K and the third technical one). One of the interests of CCE is that it can be seen as a model of the calculus of constructions with rewriting. Assuming that one uses only rewrite rules that are provable as equalities, extending conversion with $l \longrightarrow r$ can be modeled in CCE by adding $l = r$ as an axiom and then using it by extensionality. Consequently, calculus of constructions with rewriting rules which are provable as equalities is conservative with respect to the calculus of inductive constructions extended with the three axioms mentioned above.

Unfortunately we do not prove in this paper that additional rules are provable equalities. We only approach this goal by studying the notion of inductive consequences. In the algebraic setting the three notions, being an additional rule, being an inductive consequence and being an equality proved by induction coincide. In our setting the gap between these notions is not trivial: there are inductive consequences which are not provable as equalities without additional axioms, see `subst` in Section 2 where axiom K is needed, and there are additional rules that are not inductive consequences in the strict sense, see `id` in Section 2 and 7.

It would be interesting to check what happens if we try to prove equalities in a system where axiom K and functional extensionality are present from the start. Especially, since they appear independently in other works on equality in type theory (see e.g. [2]) and that one of them, axiom K, can be easily defined by rewriting, see for example [19].

References

1. Abel, A., Coquand, T., Pagano, M.: A modular type-checking algorithm for type theory with singleton types and proof irrelevance. In: Curien, P.-L. (ed.) TLCA 2009. LNCS, vol. 5608, pp. 5–19. Springer, Heidelberg (2009)
2. Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now? In: PLPV 2007, pp. 57–68. ACM, New York (2007)
3. Barbanera, F., Fernández, M., Geuvers, H.: Modularity of strong normalization in the algebraic-λ-cube. Journal of Functional Programming 7(6), 613–660 (1997)
4. Blanqui, F.: Definitions by rewriting in the Calculus of Constructions. Mathematical Structures in Computer Science 15(1), 37–92 (2005)
5. Blanqui, F., Jouannaud, J.P., Okada, M.: The calculus of algebraic constructions. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 301–316. Springer, Heidelberg (1999)
6. Blanqui, F., Jouannaud, J.P., Strub, P.Y.: Building decision procedures in the calculus of inductive constructions. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 328–342. Springer, Heidelberg (2007)
7. Chrząszcz, J.: Modules in Coq are and will be correct. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 130–146. Springer, Heidelberg (2004)
8. Chrząszcz, J.: Modules in Type Theory with Generative Definitions. PhD thesis, Warsaw University and University Paris-Sud (2004)
9. The Coq proof assistant, http://coq.inria.fr/
10. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1. Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Heidelberg (1985)
11. Hofmann, M., Streicher, T.: The groupoid model refutes uniqueness of identity proofs. In: LICS 1994, pp. 208–212. IEEE Computer Society, Los Alamitos (1994)
12. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, pp. 200–219. Springer, Heidelberg (2003)
13. Oury, N.: Extensionality in the calculus of constructions. In: Hurd, J., Melham, T.F. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 278–293. Springer, Heidelberg (2005)
14. Paulin-Mohring, C.: Inductive definitions in the system Coq: Rules and properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 328–345. Springer, Heidelberg (1993)
15. Terese (ed.): Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
16. Walukiewicz-Chrząszcz, D.: Termination of rewriting in the calculus of constructions. Journal of Functional Programming 13(2), 339–414 (2003)
17. Walukiewicz-Chrząszcz, D.: Termination of Rewriting in the Calculus of Constructions. PhD thesis, Warsaw University and University Paris-Sud (2003)
18. Walukiewicz-Chrząszcz, D., Chrząszcz, J.: Inductive consequences in the calculus of constructions, http://www.mimuw.edu.pl/~chrzaszcz/papers/
19. Walukiewicz-Chrząszcz, D., Chrząszcz, J.: Consistency and completeness of rewriting in the calculus of constructions. Logical Methods of Computer Science 4(3) (2008)
20. Werner, B.: On the strength of proof-irrelevant type theories. Logical Methods in Computer Science 4(3) (2008)

Validating QBF Invalidity in HOL4*

Tjark Weber

University of Cambridge
Computer Laboratory
tw333@cam.ac.uk

Abstract. The Quantified Boolean Formulae (QBF) solver Squolem can generate certificates of invalidity, based on Q-resolution. We present independent checking of these certificates in the HOL4 theorem prover. This enables HOL4 users to benefit from Squolem's automation for QBF problems, and provides high correctness assurances for Squolem's results. Detailed performance data shows that LCF-style certificate checking is feasible even for large QBF instances. Our work prompted improvements to HOL4's inference kernel.

1 Introduction

Deciding the validity of Quantified Boolean Formulae (QBF) is an extension of the well-known Boolean satisfiability problem (SAT). In addition to the usual connectives of propositional logic, QBF may contain universal and existential quantifiers over Boolean variables. As a simple example, consider the formula

$$\exists x \, \forall y \, \exists z. \; x \wedge (y \vee z) \wedge (y \vee \neg z). \tag{1}$$

QBF have applications in adversarial planning and formal verification [1,2,3]. They are also interesting from a theoretical viewpoint: QBF is the canonical PSPACE-complete problem [4]. Whether QBF is harder than SAT is an open problem, but it is widely believed that Boolean quantifiers allow to give exponentially more succinct encodings for certain problems than propositional logic alone.

QBF solvers automatically decide validity of such formulae. (For closed QBF, satisfiability is equivalent to validity, and unsatisfiability is equivalent to invalidity.) In addition, certain QBF solvers can produce certificates for their answers that can be checked independently [5]. Squolem is a state-of-the-art QBF solver that generates Q-resolution [6] based certificates for invalid formulae [7].

In this paper, we present independent checking of Squolem's certificates for invalid QBF in the HOL4 [8] theorem prover. HOL4 is a popular interactive theorem prover for higher-order logic [9]. It is based on a small LCF-style [10,11] kernel that provides an abstract data type of theorems, equipped with a fixed set of constructor functions (corresponding to the axiom schemata and inference rules of higher-order logic). Derived rules (such as Q-resolution) that are not

* This work was supported by the British EPSRC under grant EP/F067909/1.

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 466–480, 2010.

provided by this kernel must be implemented by composing existing rules. This provides high correctness assurances: derived rules cannot produce inconsistent theorems, as long as the theorem data type itself is implemented correctly. On the other hand, it makes an efficient implementation of derived rules challenging.

The motivation for our work is twofold. First, interactive theorem provers like Coq [12], HOL4, Isabelle [13] and PVS [14] can greatly benefit from the reasoning power of automated tools. Consequently, researchers have on various occasions integrated automated first-order provers [15,16,17], SAT solvers [18], and more recently SMT solvers [19,20] with interactive provers. Our integration of a QBF solver with HOL4 fills a small, but not insignificant gap in this long line of research. It enables HOL4 users to benefit from Squolem's automation for QBF problems, and since the results are checked by HOL4's inference kernel, no trust needs to be put in the QBF solver.

Second, QBF solvers are complex software tools. Similar to state-of-the-art SAT solvers, they typically employ various heuristics and optimizations to achieve competitive performance [21,22]. Correctness is hard to establish, and different QBF solvers frequently disagree on the status of individual benchmarks. QBF-Eval competitions in previous years resolved disagreements by majority vote [23]. This rather unsatisfactory approach confirms the importance of QBF benchmark certification. HOL4's inference kernel has been carefully scrutinized by dozens of researchers for over two decades. By using HOL4 as an independent checker, we obtain high correctness assurances for Squolem's results.

We review related work in Section 2. Relevant background material is introduced in Section 3. Section 4 presents our main contribution: an approach to QBF certificate checking, and in particular an efficient implementation of Q-resolution, in HOL4. Experimental results are given in Section 5. Section 6 concludes.

2 Related Work

To our knowledge, this paper is the first to consider the integration of a QBF solver with an interactive theorem prover. Related work can be classified into two distinct areas: (i) the integration of automated solvers with LCF-style theorem provers, and (ii) certificate checking for QBF solvers.

Integrating automated solvers with interactive theorem provers, just like our work, is typically motivated by a need for increased automation in the interactive system. First attempts were already made in the early 90s [15]. Since then, a long line of related research has developed. Integrations have been proposed for first-order provers [16,17], for model checkers [24], computer algebra systems [25,26], SAT solvers [18], and more recently for SMT solvers [19,20], to name just a few. The approach presented in this paper especially draws on ideas from [18] for efficient LCF-style propositional resolution (see Section 4).

Q-resolution based certificates for Squolem were proposed by Jussila et al. [7]. Other proof formats for QBF solvers have been suggested: e.g., BDD-based traces for sKizzo [27], and inference logs for yQuaffle [28]. Narizzano et al. [5] give an

overview and compare different certificate formats. Squolem's certificates show competitive performance, and they are relatively simple. Thus, implementing a checker is probably easier than for the other formats.

Unsurprisingly, stand-alone proof checkers for QBF are typically much more efficient [5] than the LCF-style proof checker presented in this paper. From the HOL4 point of view, however, a stand-alone checker would become part of the trusted code base (i.e., bugs in the checker—or in the integration—could lead to inconsistent theorems in HOL4). In contrast, the checker presented here cannot draw an unsound inference: any attempt to do so will be prevented by HOL4's trusted inference kernel.

3 Background and Theory

We now introduce relevant definitions and notation in more detail. Our terminology is entirely standard. The reader is expected to be familiar with propositional logic.

3.1 Quantified Boolean Formulae

We assume an infinite set of Boolean variables. A *literal* is a possibly negated Boolean variable. We extend negation to literals and identify $\neg\neg v$ with v. A *clause* is a disjunction of literals. A clause is *trivial* if it contains both a variable and its negation. We say that a propositional formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses.

Definition 1 (Quantified Boolean Formula). *A* Quantified Boolean Formula (QBF) *is of the form*

$$Q_1 x_1 \ldots Q_n x_n . \phi,$$

where $n \geq 0$, each x_i is a Boolean variable, each Q_i is either $\forall$ or $\exists$, and ϕ is a propositional formula in CNF.

$Q_1 x_1 \ldots Q_n x_n$ is called the *quantifier prefix*, and ϕ is called the *matrix*. Without loss of generality, we consider QBF in this *prenex form* only. Any formula involving only propositional connectives and quantifiers over Boolean variables can be transformed into prenex form through syntactic manipulations. (We have not yet implemented such a transformation. Note that a HOL4 implementation, aside from producing an equivalent QBF in prenex form, would also have to produce a proof of the equivalence. Doing so efficiently can easily be a challenge for large formulae, but is beyond the scope of this paper.)

We define an order $<$ on variables such that $x_i < x_j$ iff $i < j$, i.e., larger variables are in the scope of smaller variables. x_1 is called the *outermost*, x_n the *innermost* variable of the above QBF.

The QDIMACS format [29] is the standard input format of QBF solvers. It provides a textual means of encoding QBF in prenex form. It is a backward-compatible extension of the DIMACS format [30], the standard input format

of SAT solvers. We have implemented a translation from (the QBF subset of) HOL4 terms into QDIMACS format, and a simple recursive-descent parser for QDIMACS files that returns the corresponding QBF as a HOL4 term (see Section 3.4).

The QDIMACS format imposes further restrictions: all variables x_i must be distinct, all variables must appear in the matrix, and the innermost quantifier must be existential (i.e., $Q_n = \exists$). Note that an innermost universal quantifier can be eliminated by removing all occurrences of the bound variable from the matrix: if a non-trivial clause $v \vee \phi$ is true for all values of v, then ϕ must be true (and likewise for a clause $\neg v \vee \phi$). This inference is called *forall-reduction* [6]. Applying it as often as possible (to eliminate all universal variables that are larger than the largest existential variable), one obtains the *forall-reduct* of the original clause.

We further require all variables that appear in the matrix to be bound by some quantifier, i.e., we consider *closed* QBF only. This is to avoid confusion: in the QDIMACS format, free variables have existential semantics (to retain backward compatibility with the DIMACS format), while in HOL4, free variables in theorems have universal semantics (to permit instantiation). Therefore, if a QBF has free variables, we consider its existential closure instead.

The semantics of closed QBF is defined recursively, by expanding the outermost variable: $[\![\forall x.\, \phi]\!] = [\![\phi[x \mapsto \top] \wedge \phi[x \mapsto \bot]]\!]$, and similarly $[\![\exists x.\, \phi]\!] = [\![\phi[x \mapsto \top] \vee \phi[x \mapsto \bot]]\!]$. (Here $\phi[x \mapsto y]$ denotes substitution of y for all free occurrences of x in ϕ.) A QBF is called *invalid* if its semantics is $\bot$ (i.e., false).

3.2 Q-Resolution

QBF of interest typically contain several dozen or even hundreds of quantifiers. A naive recursive computation of their semantics, which would be exponential in the number of quantifiers, is not feasible. Therefore, Squolem takes a different approach. To show that a QBF is invalid, Squolem proves that it entails $\bot$. Squolem's certificates of invalidity are based on a single inference rule that is known as *Q-resolution* [6]. Q-resolution employs propositional resolution followed by forall-reduction to eliminate universal quantifiers.

Let ϕ and ψ be two clauses. We say that ϕ and ψ can be *resolved* if some variable v occurs positively in ϕ and negatively in ψ. (v is called the *pivot* variable.) Propositional resolution then derives the clause $\phi' \vee \psi'$, where ϕ' is ϕ with v removed, and ψ' is ψ with $\neg v$ removed. This clause is called the *resolvent* of ϕ and ψ.

The resolvent of non-trivial clauses no longer contains the pivot variable. If the pivot was existential, the resolvent's largest variable may be universal, thereby enabling forall-reductions.

Definition 2 (Q-resolution). *Let ϕ and ψ be two clauses of a QBF that can be resolved. Their resolvent's forall-reduct is called the* Q-resolvent *of ϕ and ψ.*

Q-resolution, just like resolution for propositional clauses, is sound and refutation-complete for QBF in prenex form [6]. Thus, given any invalid QBF, we can derive $\bot$ by repeated application of Q-resolution to suitable clauses.

As a simple example, consider (1). This QBF is invalid. To derive $\bot$ using Q-resolution, we resolve $y \vee z$ with $y \vee \neg z$ to obtain y. This clause no longer contains z, the QBF's innermost variable. Thus forall-reduction removes y, which is universally quantified, and we obtain the empty clause, i.e., $\bot$.

3.3 Squolem's Certificates of Invalidity

Squolem's certificate format is described in detail in [31]. The format is ASCII-based. Clauses and variables are referenced by positive integers. Negative values stand for negated variables, i.e., integer negation denotes propositional negation.

Certificates of invalidity contain a log of Q-resolution inferences, concluded by a final line that gives the identifier of the empty clause. For each Q-resolvent, the log contains a line stating its assigned clause identifier, the literals that the Q-resolvent contains, and the clauses that it was derived from. Original clauses (from the QBF's matrix) are numbered consecutively, starting from 1.

For instance, mapping x, $y \vee z$ and $y \vee \neg z$ to clause identifiers 1, 2 and 3, respectively, a certificate for (1) might look as follows:

```
QBCertificate
4 0 2 3 0
CONCLUDE INVALID 4
```

Thus, the empty clause (with identifier 4) is obtained by Q-resolving clauses 2 and 3. Note that forall-reduction is part of a Q-resolution inference. 0 is merely used as a separator.

We have written a simple recursive-descent parser for this certificate format that returns the encoded information as a value in Standard ML.

3.4 Higher-Order Logic

HOL4 is a popular LCF-style [10,11] theorem prover for polymorphic higher-order logic [9]. It is based on Church's simple type theory [32] extended with Hindley-Milner style polymorphism [33]. Higher-order logic (HOL) contains a type of Booleans, propositional connectives, and quantifiers over arbitrary types. Hence, quantified propositional logic can straightforwardly be embedded into HOL.

HOL4 implements a natural-deduction calculus. Theorems represent *sequents* $\Gamma \vdash \phi$, where Γ is a finite set of *hypotheses*, and ϕ is the sequent's *conclusion*. Instead of $\emptyset \vdash \phi$, we simply write $\vdash \phi$. Internally, the set of hypotheses is given by a red-black tree (for efficient search, insertion and deletion), with terms treated modulo α-equivalence.

Like other LCF-style provers, HOL4 has a small inference kernel. Theorems are implemented as an abstract data type, and new theorems can be constructed

only through a fixed set of functions provided by this data type. These functions directly correspond to the axiom schemata and inference rules of higher-order logic. Figure 1 shows the rules of HOL that we use to validate Squolem's certificates.

$$\frac{}{\{\phi\} \vdash \phi}\ \textsc{Assume} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \textsc{Conj1} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \textsc{Conj2}$$

$$\frac{\Gamma \vdash \phi \vee \psi \qquad \Delta_1 \cup \{\phi\} \vdash \theta \qquad \Delta_2 \cup \{\psi\} \vdash \theta}{\Gamma \cup \Delta_1 \cup \Delta_2 \vdash \theta}\ \textsc{DisjCases}$$

$$\frac{\Gamma \vdash \phi \implies \bot}{\Gamma \vdash \neg\phi}\ \textsc{NotIntro} \qquad \frac{\Gamma \vdash \neg\phi}{\Gamma \vdash \phi \implies \bot}\ \textsc{NotElim}$$

$$\frac{\Gamma \vdash \psi}{\Gamma \setminus \{\phi\} \vdash \phi \implies \psi}\ \textsc{Disch} \qquad \frac{\Gamma \vdash \phi \implies \psi \qquad \Delta \vdash \phi}{\Gamma \cup \Delta \vdash \psi}\ \textsc{MP}$$

$$\frac{\Gamma \vdash \phi}{\Gamma\,\theta \vdash \phi\,\theta}\ \textsc{Inst}_\theta \qquad \frac{\Gamma \vdash \forall x.\, \phi}{\Gamma \vdash \phi[x \mapsto t]}\ \textsc{Spec}_t$$

$$\frac{\Gamma \vdash \exists x.\, \phi \qquad \Delta \cup \{\phi[x \mapsto v]\} \vdash \psi}{\Gamma \cup \Delta \vdash \psi}\ \textsc{Choose}_v\ (v \text{ not free in } \Gamma,\ \Delta \text{ or } \psi)$$

Fig. 1. Selected HOL inference rules

The LCF-style architecture greatly reduces the trusted code base. Proof procedures, although they may implement arbitrarily complex algorithms, cannot produce unsound theorems, as long as the implementation of the theorem data type is correct. HOL4 is written in Standard ML [34], a type-safe functional language (with impure features, e.g., references) that has an advanced module system. To benefit from HOL4's LCF-style architecture, we must implement proof reconstruction in this language.

On top of its LCF-style inference kernel, HOL4 offers various automated proof procedures: e.g., a simplifier, which performs term rewriting, a decision procedure for propositional logic, and various first-order provers. However, the performance of these procedures is hard to control. To achieve optimal performance, we do not employ them for certificate checking, but instead combine primitive inference rules directly (see Section 4).

4 Checking Squolem's Certificates in HOL4

4.1 Preliminaries

Given a QBF $\varphi = Q_1 x_1 \ldots Q_n x_n.\, \phi$ and a certificate of its invalidity, our goal is to derive $\{\varphi\} \vdash \bot$ as a HOL4 theorem. We start by assuming the QBF's

matrix, ϕ, thereby obtaining $\{\phi\} \vdash \phi$. We then use a combination of forward and backward reasoning: the former to transform the sequent's conclusion into $\bot$, and the latter to introduce quantifiers into the hypothesis, thereby transforming it into φ. This enables a clear separation of propositional and quantifier reasoning.

Suppose that $\phi = C_1 \wedge \cdots \wedge C_k$, where each C_i is a clause of the original QBF. Repeatedly applying inference rules Conj1 and Conj2, we split $\{\phi\} \vdash \phi$ into k separate theorems $\{\phi\} \vdash C_1, \ldots, \{\phi\} \vdash C_k$. This eliminates all conjunctions from the conclusion. Therefore, we do not have to use associativity or commutativity of conjunction in order to resolve clauses. Note that this step does not consume significant amounts of memory: although ϕ may be huge, existing Standard ML implementations employ sharing and store ϕ in memory only once.

We use a similar idea to eliminate disjunctions. Suppose that $C_i = l^i_1 \vee \cdots \vee l^i_{m_i}$, where each l^i_j is a literal. Following [18], we use a combination of DisjCases, Disch and MP to transform $\{\phi\} \vdash C_i$ into $\{\phi, \neg l^i_1, \ldots, \neg l^i_{m_i}\} \vdash \bot$. This allows us to benefit from HOL4's relatively efficient management of hypotheses (which are stored in a red-black tree internally) when manipulating literals during resolution, rather than having to use associativity and commutativity of disjunction.

4.2 General Proof Structure

After transformation into this *sequent form*, each clause theorem is stored in a dictionary (implemented by a red-black tree for logarithmic time access), indexed by its numeric clause identifier. Along with each clause, we store the quantifier prefix that is missing from the clause's hypothesis. Since we started by assuming the matrix, this is the entire prefix initially, i.e., $Q_1 x_1 \ldots Q_n x_n$. During certificate validation, we will successively introduce these quantifiers again, until we arrive at the original QBF.

Squolem's certificates of invalidity encode a directed acyclic graph. The empty clause is the root, and each node is connected to the premises from which it is derived by Q-resolution. We perform a depth-first post-order traversal of this graph, starting at the root node, and adding new clauses to the clause dictionary as they are derived. This approach, which is also adopted from [18], has two benefits. First, if there are Q-resolution inferences in the certificate that do not contribute to the derivation of the final $\bot$, these are never checked in HOL4. Second, clauses must be derived in HOL4 only once, even if they are used several times in the proof. Later, they are simply retrieved from the dictionary.

4.3 Q-Resolution

Every node of the proof graph corresponds to a Q-resolution inference, which we have to perform in HOL4 (unless the node does not contribute to the derivation of the final $\bot$, see above). Q-resolution consists of propositional resolution followed by forall-reduction.

Propositional resolution for clauses in sequent form can be implemented efficiently as a combination of primitive HOL inferences [18]:

$$\dfrac{\dfrac{\Gamma \cup \{\neg v\} \vdash \bot}{\Gamma \vdash \neg v \implies \bot}\,\textsc{Disch} \qquad \dfrac{\dfrac{\Delta \cup \{v\} \vdash \bot}{\Delta \vdash v \implies \bot}\,\textsc{Disch}}{\Delta \vdash \neg v}\,\textsc{NotIntro}}{\Gamma \cup \Delta \vdash \bot}\,\textsc{MP}$$

The resolvent, $\Gamma \cup \Delta \vdash \bot$, is again a clause in sequent form.

It remains to deal with forall-reduction. Let $\{\phi, l_1, \ldots, l_m\} \vdash \bot$ be the resolvent, and let x_i be the largest variable that occurs in it. Since propositional resolution has removed the (existential) pivot variable, x_i may be universal. We must perform forall-reduction if this is the case.

There are two aspects to this task. We successively transform the QBF's matrix, which initially is a hypothesis of each clause, into the original QBF. Thus, we must introduce (existential and universal) quantifiers for variables larger than x_i, which no longer occur in the clause. Second, we must eliminate x_i, exploiting the fact that it is universal.

Suppose the missing quantifier prefix is $Q_1 x_1 \ldots \forall x_i \ldots Q_j x_j$, with $j \geq i$. If $Q_j = \forall$, we derive

$$\dfrac{\dfrac{\{\phi, l_1, \ldots, l_m\} \vdash \bot}{\{l_1, \ldots, l_m\} \vdash \phi \implies \bot}\,\textsc{Disch} \qquad \dfrac{\dfrac{}{\{\forall x_j.\, \phi\} \vdash \forall x_j.\, \phi}\,\textsc{Assume}}{\{\forall x_j.\, \phi\} \vdash \phi}\,\textsc{Spec}_{x_j}}{\{\forall x_j.\, \phi,\, l_1, \ldots, l_m\} \vdash \bot}\,\textsc{MP}$$

If $Q_j = \exists$, then necessarily $j > i$, and we instead derive

$$\dfrac{\dfrac{}{\{\exists x_j.\, \phi\} \vdash \exists x_j.\, \phi}\,\textsc{Assume} \qquad \{\phi, l_1, \ldots, l_m\} \vdash \bot}{\{\exists x_j.\, \phi,\, l_1, \ldots, l_m\} \vdash \bot}\,\textsc{Choose}_{x_j}$$

The side condition of $\textsc{Choose}_{x_j}$ (see Figure 1) is satisfied because x_j does not occur among $l_1, \ldots, l_m$.

Repeating this step for all missing quantifiers up to $Q_i x_i$, we arrive at $\{Q_i x_i \ldots Q_j x_j.\, \phi,\, l_1, \ldots, l_m\} \vdash \bot$. Note that $Q_i = \forall$, hence our reasoning is sound despite the fact that x_i still occurs in the clause.

Now x_i is bound in $Q_i x_i \ldots Q_j x_j.\, \phi$, and occurs free only in one of the literals $l_1, \ldots, l_m$. We instantiate x_i to $\neg\bot$ if it occurs positively, and to $\bot$ if it occurs negatively. In either case the literal becomes $\neg\bot$ and can be discharged.

We continue to forall-reduce the resulting clause to eliminate further universal variables if possible.

A technicality arises from the interplay of propositional resolution and forall-reduction. Forall-reduction introduces quantifiers (thereby shortening the prefix of missing quantifiers). Therefore, the two clauses used in a resolution step may have different quantifier prefixes around the matrix in their hypotheses: i.e., $Q_i x_i \ldots Q_n x_n.\, \phi$ vs. $Q_j x_j \ldots Q_n x_n.\, \phi$, with $i < j$. The resolvent will contain both formulae as hypotheses. In our bookkeeping, we only keep track of the longer prefix of missing quantifiers, i.e., $Q_1 x_1 \ldots Q_{j-1} x_{j-1}$, and ignore the

other hypothesis. Eventually, later forall-reduction steps in the proof will (again) introduce quantifiers $Q_i x_i$ through $Q_{j-1} x_{j-1}$ into the resolvent. At this point both hypotheses will become identical, and (since hypotheses are implemented by a set) one copy will automatically be discarded by HOL4.

While this could lead to an accumulation of matrix hypotheses (each with a different quantifier prefix) in a clause during certificate validation, clauses with different quantifier prefixes are rarely resolved in practice. In the QBF certificates used for evaluation (see Section 5), clauses derived by Squolem contain at most two matrix hypotheses each.

4.4 Example

Consider (1) again. Let $\phi = x \wedge (y \vee z) \wedge (y \vee \neg z)$ denote its matrix. We assume ϕ to obtain $\{\phi\} \vdash \phi$. Using Conj1 and Conj2, we derive three separate theorems $\{\phi\} \vdash x$, $\{\phi\} \vdash y \vee z$, and $\{\phi\} \vdash y \vee \neg z$. Their respective sequent forms are $\{\phi, \neg x\} \vdash \bot$, $\{\phi, \neg y, \neg z\} \vdash \bot$, and $\{\phi, \neg y, z\} \vdash \bot$ The missing quantifier prefix for each theorem is $\exists x \forall y \exists z$.

Validating Squolem's proof of invalidity (see Section 3.3), we now Q-resolve the second and the third theorem. Propositional resolution yields $\{\phi, \neg y\} \vdash \bot$. The largest variable that occurs in this clause is y.

Since y is universal, we perform forall-reduction (as detailed in Section 4.3). The innermost missing quantifier is $\exists z$. Thus, we first derive $\{\exists z.\, \phi, \neg y\} \vdash \bot$. The next missing quantifier is $\forall y$, so we derive $\{\forall y \exists z.\, \phi, \neg y\} \vdash \bot$. Now we eliminate y by instantiating it to $\bot$, thereby obtaining $\{\forall y \exists z.\, \phi, \neg \bot\} \vdash \bot$. Discharging $\neg \bot$ yields $\{\forall y \exists z.\, \phi\} \vdash \bot$. The next missing quantifier is $\exists x$, and x does not occur in the clause (except in ϕ). Hence, we finally arrive at $\{\exists x \forall y \exists z.\, \phi\} \vdash \bot$.

4.5 Variable Binding and Substitution

HOL4 comes with two different implementations of its inference kernel: one uses de Bruijn indices (and explicit substitutions) to represent λ-terms [35], the other (by M. Norrish) uses a name-carrying implementation [36]. These implementations differ in the performance (and even complexity) of primitive operations. For instance, λ-abstracting over a variable takes constant time with the name-carrying implementation, but with de Bruijn indices is linear in the size of the abstraction's body (because every occurrence of the newly bound variable in the body must be replaced by an index). Moreover, since the abstraction's body remains unchanged in the name-carrying implementation, there is more potential for memory sharing if the body is also used elsewhere, and hence a potentially smaller memory footprint. Despite these differences, both kernels show similar overall performance on the HOL4 library.

This is no longer true for QBF validation. In higher-order logic, $\forall x.\, \phi$ is syntactic sugar for $\forall(\lambda x.\, \phi)$, and likewise for $\exists x.\, \phi$. Hence, the algorithm for Q-resolution presented in Section 4.3 forms λ-abstractions (and takes them apart again) when introducing quantifiers during forall-reduction. We will see in

Section 5 that Norrish's name-carrying implementation, therefore, is significantly faster for QBF validation than the kernel that uses de Bruijn indices internally.

During evaluation, we also observed that the name-carrying implementation spent significant time instantiating variables (to $\bot$ or $\neg\bot$, before they are discharged as part of forall-reduction). Capture-avoiding substitution in a name-carrying implementation may have to rename bound variables away from the free variables in the body of a λ-abstraction. It turned out that in order to collect these free variables, the HOL4 implementation of substitution would unnecessarily descend into the body of a λ-abstraction even when the variable to be instantiated was bound (in which case instantiation would not change the body at all). We achieved an average speed-up of 2.6 (see Section 5) by improving the implementation of capture-avoiding substitution to collect free variables only when they are actually needed for renaming.

One might gain further improvements by using a modified term data structure. The kernel could compute the set of a term's free variables when the term is built, and store it in memory along with the term itself. This might allow for an even more efficient implementation of capture-avoiding substitution.

5 Experimental Results

We have evaluated our implementation on the same set of 69 invalid QBF problems that was previously used (by the Squolem authors) to evaluate the performance of Squolem's certificate generation [7]. The set resulted from applying Squolem to all 445 problems in the *2005 fixed instance* and *2006 preliminary QBF-Eval* data sets. With a time limit of 600 seconds per problem, Squolem solved 142 of these problems; 69 were determined to be invalid.

All experiments were conducted on a Linux system with an Intel Core2 Duo T9300 processor at 2.4 GHz. Memory usage was restricted to 3 GB. HOL4 was running on top of Poly/ML 5.3.

5.1 Run-Times

Table 1 shows our experimental results for the 69 invalid QBF problems. The first column gives the name of the benchmark. The next three columns provide information about the size of the benchmark, giving the number of alternating quantifiers,[1] variables, and clauses, respectively. Column four shows the run-time of Squolem (with certificate generation enabled) to solve the benchmark. Column five shows the number of Q-resolution steps in the resulting certificate. The last three columns finally show the run-time of certificate validation in HOL4, using the de Bruijn kernel, the name-carrying kernel, and our optimized variant of the name-carrying kernel, respectively (see Section 4.5). All run-times are given in seconds (rounded to the nearest tenth of a second). On one benchmark, the de Bruijn kernel ran out of memory (indicated by an M).

[1] Counting successive quantifiers of the same kind, as in $\forall x\, \forall y\, \forall z\, \ldots$, as one quantifier only. The total number of quantifiers in each benchmark is typically identical to the number of variables.

Table 1. Experimental results

Benchmark name	Quant.	Vars.	Clauses	Squolem (s)	Q-res. steps	de Bruijn (s)	name-carrying (s)	optimized name-c. (s)
Adder2-2-c	7	249	291	8.9	445	6.4	0.6	0.3
adder-2-unsat	3	66	110	13.6	3632	3.3	1.6	0.5
comp.blif_0.10_0.20_0_0_inp_exact	7	310	831	6.0	2612	3.0	0.3	0.1
comp.blif_0.10_1.00_0_0_inp_exact	3	306	842	1.9	3317	1.0	0.1	0.1
flipflop-3-c	3	164	203	0.0	15	0.0	0.0	0.0
flipflop-4-c	3	866	1158	20.5	12	1.4	0.0	0.0
k_d4_p-12	33	755	2234	0.4	276	44.2	1.5	0.6
k_d4_p-16	41	995	2966	0.6	348	95.5	2.7	1.0
k_d4_p-20	49	1235	3698	0.8	420	173.1	4.3	1.7
k_d4_p-21	51	1295	3881	0.8	438	197.4	4.7	2.0
k_d4_p-4	17	275	770	0.1	132	2.8	0.2	0.1
k_d4_p-8	25	515	1502	0.3	204	15.0	0.7	0.3
k_dum_p-12	27	517	1352	0.1	302	17.6	0.8	0.3
k_dum_p-16	35	685	1782	0.1	334	35.1	1.3	0.6
k_dum_p-20	43	853	2212	0.1	366	61.9	1.9	0.8
k_dum_p-21	45	893	2311	0.1	366	68.1	2.1	0.9
k_dum_p-4	15	556	215	0.1	231	2.2	0.2	0.1
k_dum_p-8	19	349	922	0.1	266	6.9	0.4	0.2
k_grz_p-12	17	534	1961	479.2	303	19.1	1.1	0.3
k_grz_p-4	17	318	953	0.1	283	5.3	0.4	0.2
k_grz_p-8	17	419	1406	0.6	286	10.5	0.7	0.2
k_lin_p-4	7	241	840	0.0	28	1.1	0.1	0.0
k_lin_p-8	7	439	1916	242.8	32	4.7	0.2	0.1
k_path_p-12	27	805	2238	0.3	306	40.3	1.4	0.5
k_path_p-16	35	1081	3014	0.4	406	91.9	2.6	1.0
k_path_p-20	43	1357	3790	0.5	506	173.4	4.4	1.9
k_path_p-21	45	1429	3996	0.6	531	200.2	4.9	2.1
k_path_p-4	11	253	686	0.1	106	1.7	0.1	0.1
k_path_p-8	19	529	1462	0.2	206	12.2	0.5	0.2
k_poly_p-12	79	1005	2268	0.4	1072	137.5	3.5	2.0
k_poly_p-16	103	1329	3000	1.6	1375	303.9	6.6	3.8
k_poly_p-20	127	1653	3732	1.8	1711	577.9	11.1	6.6
k_poly_p-21	131	1707	3854	0.9	1771	635.6	12.0	7.1
k_poly_p-4	31	357	804	0.1	377	7.0	0.3	0.2
k_poly_p-8	55	681	1536	0.2	724	44.1	1.5	0.7
k_t4p_p-12	37	979	3040	1.5	394	91.9	2.7	1.0
k_t4p_p-16	45	1529	3936	2.1	474	180.7	4.5	1.8
k_t4p_p-20	53	1539	4832	2.5	554	318.3	6.9	2.7
k_t4p_p-21	55	1609	5056	2.6	574	360.7	7.5	3.0
k_t4p_p-4	21	419	1248	0.5	234	9.3	0.5	0.2
k_t4p_p-8	29	699	2144	1.0	314	37.1	1.4	0.5
s27_d3_u	3	117	254	33.2	309	0.2	0.2	0.0
sat05-561-qd	1	24	61	0.0	158	0.0	0.0	0.0
TOILET2.1.iv.3	3	28	70	0.0	20	0.0	0.0	0.0
toilet_a_08_01.2	3	60	2205	1.0	6	1.7	0.8	0.1
toilet_a_08_01.4	3	112	2429	1.1	44	3.6	1.4	0.2
toilet_a_08_05.2	3	140	2833	124.9	1855	5.7	7.4	4.9
toilet_a_10_01.2	3	74	10455	33.7	6	17.8	8.0	1.0
toilet_a_10_01.3	3	106	10604	35.1	16	24.7	8.5	1.1
toilet_a_10_01.4	3	138	10753	36.3	44	36.5	13.7	1.6
toilet_c_08_01.2	3	55	229	0.0	6	0.1	0.0	0.0
toilet_c_08_01.4	3	107	453	0.1	44	0.2	0.0	0.0
toilet_c_08_05.2	3	135	857	122.4	1855	0.6	0.7	0.7
toilet_c_10_01.2	3	68	325	0.0	6	0.1	0.0	0.0
toilet_c_10_01.4	3	132	623	0.2	44	0.3	0.1	0.0
tree-exa2-10	20	20	12	0.0	18	0.0	0.0	0.0
tree-exa2-15	30	30	17	0.0	28	0.0	0.0	0.0
tree-exa2-20	40	40	22	0.0	38	0.0	0.0	0.0
tree-exa2-25	50	50	27	0.0	48	0.0	0.0	0.0
tree-exa2-30	60	60	32	0.0	58	0.1	0.0	0.0
tree-exa2-35	70	70	37	0.0	68	0.1	0.1	0.0
tree-exa2-40	80	80	42	0.0	78	0.1	0.1	0.1
tree-exa2-45	90	90	47	0.0	88	0.2	0.1	0.1
tree-exa2-50	100	100	52	0.0	98	0.3	0.2	0.1
vonNeumann-ripple-carry-5-c	3	24562	35189	220.3	33	M	1.8	1.5
z4ml.blif_0.10_0.20_0_0_inp_exact	5	65	193	1.8	996	0.2	0.0	0.0
z4ml.blif_0.10_0.20_0_0_out_exact	3	61	185	1.2	1536	1.3	0.3	0.1
z4ml.blif_0.10_1.00_0_0_inp_exact	3	65	198	0.1	588	0.1	0.0	0.0
z4ml.blif_0.10_1.00_0_0_out_exact	3	63	194	0.6	1588	1.2	0.2	0.1

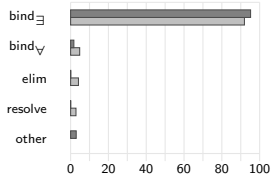


Fig. 2. de Bruijn

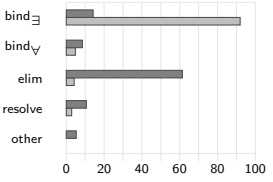


Fig. 3. Name-carrying

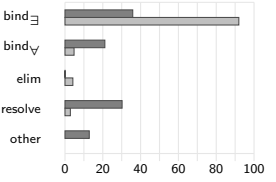


Fig. 4. Optimized

Average run-times are 60.2 seconds for the de Bruijn kernel (not including benchmark vonNeumann-ripple-carry-5-c), 2.1 seconds for the name-carrying kernel, and 0.8 seconds for our optimized variant of the name-carrying kernel. This amounts to speed-up factors of 28.7 (de Bruijn vs. name-carrying) and 2.6 (name-carrying vs. optimized), respectively, for a total speed-up factor of 75.3 (de Bruijn vs. optimized).

For comparison, we have also measured run-times of QBV [7], a stand-alone checker for Squolem's certificates that was developed by the authors of Squolem. QBV validates each of the 69 certificates in less than 0.1 seconds. LCF-style validation in HOL4, using the optimized name-carrying kernel, is one to two orders of magnitude slower. However, for users of HOL4, another comparison might be more relevant: LCF-style validation (using the optimized name-carrying kernel) on average is a factor of 24.5 faster than proof search with Squolem, and at most 8 times slower on individual benchmarks.

5.2 Profiling

To gain deeper insight into these results, we present profiling data for the de Bruijn kernel (Figure 2), the name-carrying kernel (Figure 3), and our optimized variant of the name-carrying kernel (Figure 4).

For each kernel, we show the shares of total run-time (dark bars) and relative number of function calls (light bars) for the following functions: binding of existential quantifiers during forall-reduction ($\mathsf{bind}_\exists$), binding of universal quantifiers during forall-reduction ($\mathsf{bind}_\forall$), elimination of universal variables during forall-reduction (elim), and propositional resolution (resolve) as part of Q-resolution. Additionally, time spent on other aspects of certificate validation, e.g., file parsing and conversion of clauses into sequent form, is shown as well (other). The relative number of function calls (light bars) is the same for each kernel.

We observe that the de Bruijn kernel (Figure 2) spends more than 90% of validation time on the introduction of existential quantifiers. This is in line with the relative frequency of $\mathsf{bind}_\exists$. The name-carrying implementation (Figure 3), however, performs the same operation much more quickly (for the reasons discussed in Section 4.5), reducing its run-time share to less than 20%. On the other hand, time spent on variable elimination (elim) has increased disproportionally, to over 60%. Our optimization of capture-avoiding substitution (see Section 4.5) reduces this time to a negligible fraction again (Figure 4), while the remaining operations take proportionally higher time shares.

6 Conclusions

We have presented LCF-style checking for certificates of QBF invalidity (generated by the QBF solver Squolem) in HOL4. In particular, we have presented an efficient implementation of Q-resolution on top of HOL4's inference kernel for higher-order logic. Detailed performance data shows that LCF-style certificate checking is feasible even for large invalid QBF instances: all 69 benchmark certificates were checked successfully. However, performance very much depends on implementation details of the underlying inference kernel. We have improved HOL4's implementation of capture-avoiding substitution, thereby achieving a speed-up of 75.3 over an implementation based on de Bruijn indices. Our implementation is freely available from the HOL4 repository [36].

Our work has two main applications. First, it enables HOL4 users to benefit from Squolem's automation for QBF problems. These can now be passed from the HOL4 system to Squolem, which will automatically decide their validity. For invalid QBF problems, Squolem's certificate will then be used to derive the QBF's negation as a theorem in HOL4. Second, our work provides high correctness assurances for Squolem's results. Due to HOL4's LCF-style architecture, our proof checker cannot draw unsound inferences (provided HOL4's kernel is correct). Thus, the approach can be used for QBF benchmark certification.

In this paper, we have only considered certificates of invalidity. In principle, one can establish validity of a QBF instance by showing that the negation is invalid. However, this approach is rarely feasible in practice [7]. Squolem can generate certificates of validity directly, based on Skolem functions. LCF-style checking for certificates of validity remains future work.

One could also extend our work to other QBF solvers, which use different certificate formats (see [23] for an overview), and to other interactive theorem provers, e.g., Isabelle or Coq. Because seemingly minor differences in kernel data structures can have significant impact, it is not clear if similar performance can be achieved in these systems.

An alternative approach that might yield better performance than the LCF-style implementation presented in this paper is the use of reflection [37], i.e., implementing and proving correct a checker for Squolem's certificates in the prover's logic, and then executing the verified checker without producing proofs. While this approach still provides relatively high correctness assurances, obtaining a theorem in HOL4 would require enhancing the inference kernel with a reflection rule that allows to trust the result of such a verified computation.

Acknowledgments

The author would like to thank Christoph Wintersteiger for answering various questions about Squolem.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 401–413. Springer, Heidelberg (2004)
3. Hanna, Z., Dershowitz, N., Katz, J.: Bounded model checking with QBF. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 408–414. Springer, Heidelberg (2005)
4. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: Proc. 5th Annual ACM Symp. on Theory of Computing, pp. 1–9 (1973)
5. Narizzano, M., Peschiera, C., Pulina, L., Tacchella, A.: Evaluating and certifying QBFs: A comparison of state-of-the-art tools. AI Communications 22(4), 191–210 (2009)
6. Büning, H.K., Karpinski, M., Flögel, A.: Resolution for quantified boolean formulas. Information and Computation 117(1), 12–18 (1995)
7. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 201–214. Springer, Heidelberg (2007)
8. Slind, K., Norrish, M.: A brief overview of HOL4. In: [38], pp. 28–32
9. Gordon, M.J.C., Pitts, A.M.: The HOL logic and system. In: Towards Verified Systems. Real-Time Safety Critical Systems Series, vol. 2, pp. 49–70. Elsevier, Amsterdam (1994)
10. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
11. Gordon, M.: From LCF to HOL: a short history. In: Proof, language, and interaction: essays in honour of Robin Milner, pp. 169–185. MIT Press, Cambridge (2000)
12. Bertot, Y.: A short presentation of Coq. In: [38], pp. 12–16
13. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: [38], pp. 33–38
14. Owre, S., Shankar, N.: A brief overview of PVS. In: [38], pp. 22–27
15. Kumar, R., Kropf, T., Schneider, K.: Integrating a first-order automatic prover in the HOL environment. In: Archer, M., Joyce, J.J., Levitt, K.N., Windley, P.J. (eds.) Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, pp. 170–176. IEEE Computer Society, Los Alamitos (1992)
16. Hurd, J.: An LCF-style interface between HOL and first-order logic. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 134–138. Springer, Heidelberg (2002)
17. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)
18. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. Journal of Applied Logic 7(1), 26–40 (2009)
19. Ge, Y., Barrett, C.: Proof translation and SMT-LIB benchmark certification: A preliminary report. In: 6th International Workshop on Satisfiability Modulo Theories, SMT 2008 (2008)
20. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3, To appear at the International Conference on Interactive Theorem Proving, ITP-10 (2010)

21. Letz, R.: Lemma and model caching in decision procedures for quantified boolean formulas. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 5–15. Springer, Heidelberg (2002)
22. Pulina, L., Tacchella, A.: Learning to integrate deduction and search in reasoning about quantified boolean formulas. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 350–365. Springer, Heidelberg (2009)
23. Narizzano, M., Pulina, L., Tacchella, A.: Report of the third QBF solvers evaluation. JSAT 2(1-4), 145–164 (2006)
24. Amjad, H.: Combining model checking and theorem proving. Technical Report UCAM-CL-TR-601, University of Cambridge Computer Laboratory, Ph.D. Thesis (2004)
25. Ballarin, C.: Computer algebra and theorem proving. Technical Report UCAM-CL-TR-473, University of Cambridge Computer Laboratory, Ph.D. Thesis (1999)
26. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining coq and gappa for certifying floating-point programs. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) Calculemus 2009. LNCS, vol. 5625, pp. 59–74. Springer, Heidelberg (2009)
27. Benedetti, M.: sKizzo: A suite to evaluate and certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
28. Yu, Y., Malik, S.: Validating the result of a quantified boolean formula (QBF) solver: theory and practice. In: Tang, T. (ed.) Proceedings of the 2005 Conference on Asia South Pacific Design Automation, ASP-DAC 2005, Shanghai, China, January 18-21, pp. 1047–1051. ACM Press, New York (2005)
29. QDIMACS standard version 1.1 (2005) (released on December 21, 2005), http://www.qbflib.org/qdimacs.html (retrieved January 22, 2010)
30. DIMACS satisfiability suggested format (1993) ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc (retrieved January 22, 2010)
31. Kroening, D., Wintersteiger, C.M.: A file format for QBF certificates (2007), http://www.verify.ethz.ch/qbv/download/qbcformat.pdf (retrieved September 20, 2009)
32. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5, 56–68 (1940)
33. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL, pp. 207–212 (1982)
34. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML–Revised. MIT Press, Cambridge (1997)
35. Barras, B.: Programming and computing in HOL. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 17–37. Springer, Heidelberg (2000)
36. HOL4 contributors: HOL4 Kananaskis 5 source code (2010), http://hol.sourceforge.net/ (retrieved January 22, 2010)
37. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge (1995), http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz (retrieved April 8, 2010)
38. Mohamed, O.A., Muñoz, C., Tahar, S. (eds.): TPHOLs 2008. LNCS, vol. 5170. Springer, Heidelberg (2008)

Higher-Order Abstract Syntax in Isabelle/HOL

Douglas J. Howe

Carleton University, Ottawa ON K1S 5B6, Canada
howe@scs.carleton.ca

Abstract. *Higher Order Abstract Syntax*, or *HOAS*, is a technique for using a higher-order logic as a metalanguage for an object language with binding operators. It avoids formalizing syntactic details related to variable binding by identifying variables of the object logic with variables of the metalogic. In another paper we extended the usual set-theoretic semantics of HOL with a notion of *parametric* function, and showed how to use this extension to give solutions to the recursive type equations characteristic of HOAS, for example $T = T \times T + T \to T$ for a HOAS representation of the untyped lambda-calculus. This paper describes an effort to apply these semantic ideas in a proof assistant.

When formalizing abstract syntax that has binding constructs, one can use a first-order style with explicit representation of variable names, or a *higher-order* style [1], where variables are not explicitly represented, but instead are identified with variables of the metalanguage (HOL in our case).

The distinction can be illustrated using recursive type equations and, as an example, the syntax of untyped lambda calculus. Taking *id* to be a type of identifiers, the syntax of the untyped lambda calculus could be formalized as a datatype

$$exp = id + exp \times exp + id \times exp.$$

The first disjunct of the disjoint sum above represents variables, the second, application terms, and the third, lambda abstractions. Contrast this with the following "datatype", which uses a *higher-order* encoding:

$$exp = exp \times exp + exp \to exp.$$

Here variables are not explicitly represented: a lambda abstraction $\lambda x.\, M$ is represented as a function that can be thought of as mapping terms N to the result of substituting N for x in M.

A drawback to the HOAS approach is that it has been difficult to adapt to widely-used general-purpose proof construction tools such as Isabelle/HOL [3]. Part of the reason can be seen from the second equation above: the occurrence of *exp* on the left side of the function space arrow means the equation cannot be viewed as an inductive definition, at least when function space is given its usual meaning.

However, as shown in [2], the second equation does make sense, as an inductive definition, under a nonstandard interpretation of the function space as a type of

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 481–484, 2010.

parametric functions. These are functions that do not analyse, or discriminate on, their input, but instead just build data structures that have the input as a component. For example, $\lambda x.\,((x, 3+4),\ \mathit{Inl}\ x)$ is parametric since the body builds on x using only data constructors.

The goal of our work is to make the HOAS technique available in a general and simple way in general-purpose proof tools. In [2], we gave an extension to classical higher-order logic, and its standard set-theoretic semantics, that supports both a parametric function type, and a recursive type constructor similar to what is implicit in Isabelle/HOL's datatype package except that it also allows some negative occurrences of the type being defined.

Our previous paper focused on semantics and typechecking. The paper suggested some axioms, but mostly it just noted that a few forms of reasoning, such as HOAS-style induction, were valid in the semantics. One major shortcoming was a convincing method for dealing with representation of *open* terms. The usual HOAS-style datatypes only represent closed terms.

We are now in the process of implementing these ideas and experimenting with their use in formal metareasoning. This paper describes a first, but substantial, step in this direction. Specifically, we give an axiomatic extension of HOL for parametric functions and for *contexts*, which are iterations of parametric lambda-abstraction. Contexts represent open terms without sacrificing the identification of object-level and meta-level variables.

In the remainder of the paper we sketch our implementation. The Isabelle theories can be found at `www.scs.carleton.ca/~howe/shoas/itp10`. For a comparison with related work, see [2].

The starting point is the parametric function type. In [2], we extended HOL's type system so that parametricity of functions is guaranteed by typechecking. Modifying Isabelle's type system to implement this is not feasible, so we take the parametric function space $\alpha \hookrightarrow \beta$ to be a subset of the usual function space $\alpha \to \beta$, using a predicate *parm*. We use the notations $\sqcap x.\,e$ and $f\,\$\,e$ for the parametric versions of lambda-abstraction and application. These notations stand for $\mathit{pfun}(\lambda x.\,e)$ and $\mathit{papp}\ f\ e$, respectively. We add axioms asserting the parametricity of the identity function; the "primitive" data constructors for sum, product and natural numbers; and the composition of parametric functions.

The crux of our approach to implementing parametricity-based metareasoning is contexts. The intention of the type $\alpha \lhd \beta$ is to collect together all types of the form $\alpha \hookrightarrow \alpha \hookrightarrow \ldots \hookrightarrow \alpha \hookrightarrow \beta$ for zero or more repetitions of α. While $\alpha \hookrightarrow \beta$ is the type of one-argument parametric functions, $\alpha \lhd \beta$ can be thought of as the type of n-ary parametric functions, for all n. An n-ary parametric function, viewed as a context, represents an expression of the object logic with n free variables.

We want a datatype definition such as

$$\alpha \lhd \beta \ =\ \mathit{Closed}\ \beta \ \mid \mathit{Bind}\ \alpha \hookrightarrow \beta$$

for representing open terms of an object logic. An object $\mathit{Closed}\ x$ represents a closed term of the object logic, while *Bind* adds a "free" variable to the context. Our justification for the use of $\hookrightarrow$ in an inductive datatype definition is

semantic, so we need to axiomatize this type. This involves declaring *Closed* and *Bind* as new constants and giving axioms corresponding to the usual theorems proved for new datatypes (e.g. induction, and injectiveness and distinctness of constructors). We also add a constant for the usual primitive recursion operation over a datatype, and axiomatize its behaviour.

These axioms are not sufficient for our use of contexts, since they are only what we would get if we made a conventional datatype definition using $\rightarrow$ instead of $\hookrightarrow$. The operations we want on contexts could all be defined, instead of axiomatized, if we were not concerned with parametricity. However, what we want is to "lift" a parametric function to contexts: if f has type $\alpha \hookrightarrow \beta$ and $\sqcap x_1. \ldots \sqcap x_n. e$ is a context (so e is parametric in $x_1, \ldots, x_n$), then $\sqcap x_1. \ldots \sqcap x_n. f\,\$\,e$ is also a context. While there is an axiom for ordinary composition of parametric functions, there is no way to use it to prove this more general version of composition, so we axiomatize it. We introduce two constants

$$\begin{aligned} &\mathit{pfunc} :: (\alpha \lhd \beta) \rightarrow \alpha \lhd (\alpha \hookrightarrow \beta) \\ &\mathit{pappc} :: (\alpha \lhd (\beta \hookrightarrow \gamma)) \rightarrow (\alpha \lhd \beta) \rightarrow \alpha \lhd \gamma \end{aligned}$$

and axiomatize structural recursion for them. The constant *pappc* captures the more general composition, while *pfunc* uses *pfun* to move the last "free variable" of the context into the *Closed* part.

One might suspect that the *parm* predicate might be an issue in proofs. It has not been a problem so far: some simple lemmas provided to the simplifier have sufficed. However, no approach to reasoning about binding has turned out to be perfect, and ours is no exception. While alpha-equivalence is completely handled by the metatheory, there is another equivalence that arises with open terms. Consider our lambda-calculus example and the two contexts $\sqcap x.\, \mathit{App}\ x\ x$ and $\sqcap x.\ \sqcap y.\, \mathit{App}\ x\ x$, where we use the more suggestive *App* instead of the left injection into the disjoint sum. Both of these contexts represent the same term of the lambda calculus: $x(x)$ for some free variable x. Context equivalence is therefore defined by $c_1 \sim\!=\!\sim c_2$, for c_1 and c_2 in $\alpha \lhd \beta$, if for all l, $\mathit{ctxt\_inst}\ c_1\ l = \mathit{ctxt\_inst}\ c_2\ l$, where $\mathit{ctxt\_inst}\ c\ l$ applies c, qua iterated abstraction, to as many as needed of the infinite list of arguments l, obtaining a value of type β . We add an axiom asserting that *pfunc* respects this equivalence.

The final thing we need to axiomatize for contexts is a well-founded ordering, denoted $\preceq$. This ordering is the key to proving induction principles for HOAS representations. The ordering is based on the structural ordering of data built from pairing and disjoint sum injections, but lifted to contexts. In addition to axioms for transitivity and well-foundedness, we have axioms

$$x \preceq \mathit{pairc}\ x\ y; \quad y \preceq \mathit{pairc}\ x\ y; \quad x \preceq \mathit{inlc}\ x; \quad x \preceq \mathit{inrc}\ x; \quad x \preceq \mathit{pfunc}\ x$$

where *pairc*, *inlc* etc are defined in terms of the corresponding data constructors, for example $\mathit{inlc}\ c = \mathit{pappc}\,(\mathit{Closed}\,(\mathit{pfun}\ \mathit{Inl}))c$. The last of these axioms is noteworthy. It says, for example, that $\mathit{Bind}\,(\sqcap x.\ \mathit{Closed}\ h\,\$\,x) \preceq \mathit{Closed}\ h$. In terms of HOAS representations, this means that moving an abstraction from a term representation into the context takes us lower in the ordering. Finally, we need

an axiom for the "base case": $\mathit{is\_var}\ y \Rightarrow \neg(x \preceq y)$ where $\mathit{is\_var}\ y$ is defined to be true exactly when y has the form $\sqcap x_1.\ \ldots \sqcap x_n.\ x_i$ for some i with $1 \leq i \leq n$.

If we were able to specify all the data constructors at once, we would need no further axioms. However, for each new recursive type we introduce, we need five axioms related to the isomorphism between the left and right hand sides of the equation. We do not need axioms for the constructors of the new type because they are defined in terms of pairing and injections.

With the given axioms, we are able to prove some useful lemmas and theorems. The highlights of what we have proven so far include two forms of induction for the HOAS representation of the untyped lambda calculus. The representing type is axiomatized using the isomorphism pair consisting of *Fold_exp*, which has type $(\mathit{exp} \times \mathit{exp} + \mathit{exp} \hookrightarrow \mathit{exp}) \hookrightarrow \mathit{exp}$, and its inverse *unfold_exp*.

The context-based and HOAS induction principles (both proven) are as follows. In the first rule, the quantifications are over $\mathit{exp} \lhd \mathit{exp}$ and in the second, *exp*.

$$(\forall x.\ \mathit{is\_var}\ x \Rightarrow p\ x)\ \ \&\ \ (\forall x, y.\ p\ x \Rightarrow p\ y \Rightarrow p\ (\mathit{appc}\ x\ y))\ \ \&$$
$$(\forall x.\ p\ x \Rightarrow p\ (\mathit{lamc}\ x))\ \ \Rightarrow\ \ p\ x$$

$$(\forall x, y.\ p\ x \Rightarrow p\ y \Rightarrow p\ (\mathit{App}\ x\ y))\ \ \&$$
$$(\forall f.\ (\forall x.\ p\ x \Rightarrow p\ (f\ x)) \Rightarrow p\ (\mathit{Lam}\ f))$$
$$\Rightarrow p\ x$$

Future Work. The next steps are to implement primitive recursion operations corresponding to the two kinds of induction rules presented above, to develop some basic metamathematics of the untyped and typed lambda calculus, and to investigate implementing a "package" for the new kind of recursive data type.

Definitional developments are, of course, highly preferable to axiomatic ones such as this one, which are prone to errors (indeed, one of the reviewers of this paper found an egregious one). However, further experimentation with this approach is needed before deciding whether it merits the effort involved in a definitional construction. Because this approach is based on a change to the basic semantics of HOL, such an effort would likely be massive.

References

1. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery 40(1), 143–184 (1993)
2. Howe, D.J.: Higher-order abstract syntax in classical higher-order logic. In: Proceedings of the 4th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009) (2009)
3. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

Separation Logic Adapted for Proofs by Rewriting

Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. We present a formalisation of separation logic which, by avoiding the use of existential quantifiers, allows proofs that only use standard equational rewriting methods as found in off-the-shelf theorem provers. This proof automation is sufficiently strong to free the user from dealing with low-level details in proofs of functional correctness. The work presented here has been implemented in HOL4 and ACL2. It is illustrated on a standard example (reversal of a linked-list).

1 Introduction

Separation logic [7] has emerged as an effective technique for proving the correctness of pointer manipulating programs. As a result, there have been a number of theorem prover formalisations of separation logic [1,4,9,11] and tactics for dealing with separation logic-style reasoning in theorem provers [2,5,10].

In this paper we present a novel formalisation which, by avoiding the use of existential quantifiers, allows basic rewriting to suffice as a useful proof tactic. We believe that the simplicity of our setup makes it independent of any particular theorem prover: the work presented here has been implemented in HOL4 [8] (by the current author) and subsequently in ACL2 [3] (by Matt Kaufmann).

Gast [2] also modifies separation logic to fit better with current theorem provers. Compared with Gast [2], our approach follows the idea of separation logic more closely (we state memory content and memory layout together) and achieves a higher degree of automation by simple rewriting (Sections 3 and 4).

2 Separation Logic without Quantifiers

At the heart of separation logic lies the separating conjunction $*$. The separating conjunction is defined such that $p * q$ holds for state s whenever state s can be split into two disjoint parts (according to some definition of disjoint union $\uplus$) such that p holds for one part and q for the other part.

$$(p * q)\ s \quad = \quad \exists s_1\ s_2.\ (s = s_1 \uplus s_2) \wedge p\ s_1 \wedge q\ s_2$$

The value of the separating conjunction becomes apparent in the context of (avoiding unwanted) pointer aliasing: if $\mathsf{list}\ a\ xs$ asserts that a linked list is in memory then $\mathsf{list}\ a\ xs * \mathsf{list}\ b\ ys$ states that there are two list in memory

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 485–489, 2010.

and these occupy disjoint addresses in memory. Thus asserting that there is no pointer aliasing between the lists. This list assertion is conventionally defined as follows. Here the notation $a \mapsto x$ means that memory location a holds value x.

$$\begin{array}{lcl} \mathsf{list}\ a\ [] & = & (a = 0) \\ \mathsf{list}\ a\ (x{::}xs) & = & \exists a'.\ (a \mapsto a') * (a{+}1 \mapsto x) * \mathsf{list}\ a'\ xs * (a \neq 0) \end{array}$$

In this paper, we will show that the common case of linked-lists and other assertions of the form $(a \mapsto x) * (b \mapsto y) * \ldots * (c \mapsto z)$ can be formalised without quantifiers in such a way that simple rewriting is a sufficiently powerful proof tool for proofs of functional correctness.

Instead of defining a separating conjunction directly, we define a function separate which mimics $(a \mapsto x) * (b \mapsto y) * \ldots * (c \mapsto z)$ when supplied with a list of the form $[(a, x), (b, y), \ldots, (c, z)]$ and here the separation is due to all_distinct xs, which states that there are no duplicate elements in its argument list xs. Below t is a list of addresses that must be distinct from those mentioned in l.

$$\begin{array}{lcl} \mathsf{separate}\ []\ t\ state & = & \mathsf{all\_distinct}\ t \\ \mathsf{separate}\ ((a, x){::}l)\ t\ state & = & (state(a) = x) \wedge \mathsf{separate}\ l\ (a{::}t)\ state \end{array}$$

A suitable adaption of list is then defined as a function listx which produces a list of (address,value) pairs which can be substituted for the variable l above. To avoid the existential quantifier used in the definition of list, we make the internal addresses external and explicit. Here addr xs returns a pointer to the head of the list xs.

$$\begin{array}{lcl} \mathsf{addr}\ [] & = & 0 \\ \mathsf{addr}\ ((a, x){::}xs) & = & a \\ \mathsf{listx}\ [] & = & [] \\ \mathsf{listx}\ ((a, x){::}xs) & = & (a, \mathsf{addr}\ xs){::}(a{+}1, x){::}\mathsf{listx}\ xs \end{array}$$

Separation logic states the correctness of programs as Hoare triple judgments $\{pre\}\ code\ \{post\}$. We define a similar judgement as follows based on an operational semantics exec (defined in [6]) for a toy machine language where the code resides in memory. This judgement spec is defined to assert that, if the initial state s satisfies pre_1 and contains the code $code$ separately from addresses pre_2, then n steps of execution will reach a state which satisfies $post_1$ and contains $code$ separately from addresses $post_2$. We write list append as $+\!+$.

$$\begin{array}{l} \mathsf{spec}\ s\ n\ (pre_1, pre_2)\ code\ (post_1, post_2)\ = \\ \quad \mathsf{separate}\ (code\ +\!+\ pre_1)\ pre_2\ s\ \Longrightarrow \\ \quad \mathsf{separate}\ (code\ +\!+\ post_1)\ post_2\ (\mathsf{exec}\ n\ s) \end{array}$$

3 Automatic Rewriting Tactic

The above definitions allow rewriting alone to suffice for proving specifications. Our rewriting tactic essentially just expands all the definitions and rewrites

where applicable with lemmas that describe append (++) i.e. the expansion of separate $(xs \mathbin{+\!\!+} ys)\ t\ s$ and all_distinct $(xs \mathbin{+\!\!+} ys)$.

This simple rewriting tactic is capable of proving specifications for single passes though code. For example, it can automatically prove the following specification for a sequence of four instructions which swap the next-pointers in two linked lists, thus swapping the tails of the lists (xs is swapped for ys). The program counter which is incremented by 12 is stored at address 0. This specification states that addresses 3 and 4 are used as temporaries during execution. Their initial and final values are not recorded. Let $\mathsf{llist}\ p\ xs = (p, \mathsf{addr}\ xs)::\mathsf{listx}\ xs$.

$$\begin{array}{l}\mathsf{spec}\ s\ 4\\ \quad([(0,p)] \mathbin{+\!\!+} \mathsf{llist}\ 1\ (x::xs) \mathbin{+\!\!+} \mathsf{llist}\ 2\ (y::ys) \mathbin{+\!\!+} frame, [3,4] \mathbin{+\!\!+} rest)\\ \quad(\mathsf{pointer\_swap\_code}\ p)\\ \quad([(0,p{+}12)] \mathbin{+\!\!+} \mathsf{llist}\ 1\ (x::ys) \mathbin{+\!\!+} \mathsf{llist}\ 2\ (y::xs) \mathbin{+\!\!+} frame, [3,4] \mathbin{+\!\!+} rest)\end{array}$$

The reason for why rewriting alone can prove this is very simple: the expansion of preconditions produces a number of inequalities, e.g. $p \neq q$, on the left-hand-side of the implication in spec. These inequalities resolve if-statements,

$$\mathsf{if}\ p = q\ \mathsf{then}\ x\ \mathsf{else}\ state(p)$$

that arise in the expansion of postconditions, i.e. the right hand-side of the implication in the definition of spec.

4 Verification Example

Finally, we will demonstrate the use of our rewriting tactic as part of a standard example: verification of in-place list reversal.

The code we will verify expects on entry, that location 1 holds a pointer to the linked-list which is to be reversed. On each iteration of the loop (around location 18), one element of the list is popped from the list pointed to from location 1 and prepended to a list whose pointer is kept in location 2. On exit, location 2 holds a pointer to the reversed list. Our toy machine language has no registers, thus locations 1, 2 and 3 are used here as if they were registers.

```
 0 :  mem[2] := 0
 3 :  jump to 18
 6 :  mem[3] := mem[mem[1]]
 9 :  mem[mem[1]] := mem[2]
12 :  mem[2] := mem[1]
15 :  mem[1] := mem[3]
18 :  jump to 6, if mem[1] ≠ 0
```

We first prove a lemma about the behaviour of the body of the loop. The loop body transfers an element from one of the linked lists to the other, looping around program location 18 in the code rev_code, which is positioned relative to address p. This goal is automatically discharged by our rewriting tactic.

$$\begin{array}{l}\mathsf{spec}\ s\ 5\\ \quad([(0,p{+}18)] +\!\!+ \mathsf{llist}\ 1\ (x{::}xs) +\!\!+ \mathsf{llist}\ 2\ ys +\!\!+ \mathit{frame}, [3] +\!\!+ \mathit{rest})\\ \quad(\mathsf{rev\_code}\ p)\\ \quad([(0,p{+}18)] +\!\!+ \mathsf{llist}\ 1\ xs +\!\!+ \mathsf{llist}\ 2\ (x{::}ys) +\!\!+ \mathit{frame}, [3] +\!\!+ \mathit{rest})\end{array}$$

The proof of the main loop is a simple induction on the length of the list pointed to from location 1, i.e. xs. The base case is solved by our rewrite tactic; the step case is a simple 4-line proof which composes the above lemma for the body of the loop with the inductive hypothesis.

$$\begin{array}{l}\mathsf{spec}\ s\ (1 + \mathsf{length}\ xs \times 5)\\ \quad([(0,p{+}18)] +\!\!+ \mathsf{llist}\ 1\ xs +\!\!+ \mathsf{llist}\ 2\ ys +\!\!+ \mathit{frame}, [3] +\!\!+ \mathit{rest})\\ \quad(\mathsf{rev\_code}\ p)\\ \quad([(0,p{+}21)] +\!\!+ \mathsf{llist}\ 2\ (\mathsf{reverse}\ xs +\!\!+ ys) +\!\!+ \mathit{frame}, [1,3] +\!\!+ \mathit{rest})\end{array}$$

By joining the above specification with a similar lemma for the initialisation code, we arrive at the final specification which states that list xs is reversed:

$$\begin{array}{l}\mathsf{spec}\ s\ (3 + \mathsf{length}\ xs \times 5)\\ \quad([(0,p)] +\!\!+ \mathsf{llist}\ 1\ xs +\!\!+ \mathit{frame}, [2,3] +\!\!+ \mathit{rest})\\ \quad(\mathsf{rev\_code}\ p)\\ \quad([(0,p{+}21)] +\!\!+ \mathsf{llist}\ 2\ (\mathsf{reverse}\ xs) +\!\!+ \mathit{frame}, [1,3] +\!\!+ \mathit{rest})\end{array}$$

Acknowledgements. I would like to thank Matt Kaufmann for writing and explaining how this work can be reproduced in the ACL2 theorem prover. His ACL2 script, which includes informative comments, is available on-line [6]. This work was partially supported by EPSRC Research Grant EP/G007411/1.

References

1. 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)
2. Gast, H.: Lightweight separation. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 199–214. Springer, Heidelberg (2008)
3. Kaufmann, M., Moore, J.S.: An ACL2 tutorial. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 17–21. Springer, Heidelberg (2008)
4. Marti, N., Aeldt, R., Yonezawa, A.: Towards formal verification of memory properties using separation logic. In: Workshop of the Japan Society for Software Science and Technology, Japan Society for Software Science and Technology, Japan (2005)
5. McCreight, A.: Practical tactics for separation logic. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 343–358. Springer, Heidelberg (2009)
6. Myreen, M.O., Kaufmann, M.: HOL4 and ACL2 implementations, HOL4 (Myreen): ACL2 (Kaufmann), http://www.cl.cam.ac.uk/~mom22/sep-rewrite/
7. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proceedings of Logic in Computer Science (LICS). IEEE Computer Society, Los Alamitos (2002)

8. 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)
9. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Principles of Programming Languages (POPL), pp. 97–108. ACM, New York (2007)
10. Tuerk, T.: A formalisation of smallfoot in HOL. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 469–484. Springer, Heidelberg (2009)
11. Weber, T.: Towards mechanized program verification with separation logic. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 250–264. Springer, Heidelberg (2004)

Developing the Algebraic Hierarchy with Type Classes in Coq

Bas Spitters and Eelis van der Weegen

Radboud University Nijmegen

Abstract. We present a new formalization of the algebraic hierarchy in Coq, exploiting its new type class mechanism to make practical a solution formerly thought infeasible. Our approach addresses both traditional challenges as well as new ones resulting from our ambition to build upon this development a library of constructive analysis in which abstraction penalties inhibiting efficient computation are reduced to a bare minimum. To support mathematically sound abstract interfaces for $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$, our formalization includes portions of category theory and multisorted universal algebra.

1 Introduction

The development of libraries for formalized mathematics presents many software engineering challenges [4,8], because it is far from obvious how the clean, idealized concepts from everyday mathematics should be represented using the facilitities provided by concrete theorem provers and their formalisms, in a way that is both mathematically faithful and convenient to work with.

For the algebraic hierarchy—a critical component in any library of formalized mathematics—these challenges include structure inference, handling of multiple inheritance, idiomatic use of notations, and convenient algebraic manipulation.

Several solutions have been proposed for the Coq theorem prover: dependent records [7] (a.k.a. telescopes), packed classes [6], and occasionally modules. We present a new approach based entirely on Coq's new type class mechanism, and show how its features together with a key design pattern let us effectively address the challenges mentioned above.

Since we intend to use this development as a basis for constructive analysis with practical certified exact real arithmetic, an additional objective and motivation in our design is to facilitate *efficient* computation. In particular, we want to be able to effortlessly swap implementations of number representations. Doing this requires that we have clean abstract interfaces, and mathematics tells us what these should look like: we represent $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ as *interfaces* specifying an initial semiring, an initial ring, and a field of integral fractions, respectively.

To express these elegantly and without duplication, our development[1] includes an integrated formalization of parts of category theory and multi-sorted universal algebra, all expressed using type classes for optimum effect.

[1] The sources are available at: `http://www.eelis.net/research/math-classes/`

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 490–493, 2010.

2 The Type-Classified Algebraic Hierarchy

Unlike Haskell's and Isabelle's second class type classes, Coq's type classes are first class: classes and their instances are realized as ordinary record types ("dictionaries") and registered constants of these types.

We represent each structure in the algebraic hierarchy as a type class. This immediately leads to the familiar question of which components of the structure should become parameters of the class, and which should become fields. By far the most important design choice in our development is the decision to turn all *structural* components (i.e. carriers, relations, and operations) into parameters, keeping only *properties* as fields. Type classes defined in this way are essentially predicates with automatically resolved proofs.

Conventional wisdom warns that while this approach is theoretically very flexible, one risks extreme inconvenience both in having to declare and pass around all these structural components all the time, as well as in losing notations (because we no longer project named operations out of records).

These are legitimate concerns that we avoid by exploiting the way Coq type classes and their support infrastructure work, using *operational type classes*: classes with a single field representing a single relation or operation in isolation. Such classes are treated specially by Coq in being translated to mere definitions rather than records, with the field projection becoming the identity function.

Class *Equiv A* := *equiv* : *relation A*.
Infix `"="` := *equiv* (*at level 70*, *no associativity*) .

These operational type classes serve to establish *canonical names*, which not only lets us bind notations to them, but also makes their declaration and use implicit in most contexts. For instance, using the following definition of semirings, all structural parameters (represented by operational classes declared with curly brackets) will be implicitly resolved by the type class mechanism rather than listed explicitly whenever we talk about semirings.

Class *SemiRing A* {*e* : *Equiv A*} {*plus* : *RingPlus A*} {*mult* : *RingMult A*}
{*zero* : *RingZero A*} {*one* : *RingOne A*} : *Prop* :=
{*semiring_mult_monoid* :> *Monoid A* (*op* := *mult*) (*unit* := *one*)
; *semiring_plus_monoid* :> *Monoid A* (*op* := *plus*) (*unit* := *zero*)
; *semiring_plus_comm* :> *Commutative plus*
; *semiring_mult_comm* :> *Commutative mult*
; *semiring_distr* :> *Distribute mult plus*
; *mult_0_l* : $\forall\ x, 0 * x\ =\ 0$} .

The two key Coq features that make this work are implicit quantification (when declaring a semiring), and maximally inserted implicit arguments (when stating that something is a semiring, and when referencing operations and relations). Both were added specifically to support type classes.

Having argued that the *all-structure-as-parameters* approach *can* be made practical, we enumerate some of the benefits that make it worthwhile.

First, multiple inheritance becomes trivial: *SemiRing* inherits two *Monoid* structures on the same carrier and setoid relation, using ordinary named arguments (rather than dedicated extensions [9]) to achieve "manifest fields".

Second, because our terms are small and independent and never refer to proofs, we are invulnerable to concerns about efficiency and ambiguity of projection paths that plague existing solutions, obviating the need for extensions like the proposed coercion pullbacks [1].

Third, since our structural type classes are mere predicates, overlap between their instances is a non-issue. Together with the previous point, this gives us tremendous freedom to posit multiple structures on the same operations and relations, including ones derived implicitly via subclasses: by simply declaring a *SemiRing* class instance showing that a ring is a semiring, results about semirings immediately apply implicitly to any known ring, without us having to explicitly encode this relation in the hierarchy definition itself, and without needing any projection or translation of carriers or operations.

3 Category Theory and Universal Algebra

Motivated originally by our desire to cleanly express interfaces for basic numeric data types such as $\mathbb{N}$ and $\mathbb{Z}$ in terms of their categorical characterization as initial objects in the categories of semirings and rings, respectively, we initially introduced only the very basics of category theory into our development, again using type classes where possible to achieve the same benefits mentioned before.

Realizing that much code duplication for the various algebraic structures in the hierarchy could be avoided by employing universal algebra constructions, we then proceeded to formalize some of the theory of multisorted universal algebra and equational theories, using it to automatically construct varieties of algebras. We avoided existing formalizations [3,5] of universal algebra, because we aimed to find out what level of elegance, convenience, and integration can be achieved using the state of the art technology (of which type classes are the most important instance).

At the time of writing, our development includes a fully integrated formalization of a nontrivial portion of category theory and multisorted universal algebra, including various categories (e.g. the category *Cat* of categories, and generic variety categories which we instantiate to obtain the categories of monoids, semirings, and rings), functors (including automatically generated forgetful functors), natural transformations, adjunctions, initial models of equational theories constructed from term algebras, transference of proofs between isomorphic models of equational theories, subalgebras, congruences, quotients, products, and the first homomorphism theorem.

There is an interesting interplay in our development between concrete algebraic structure type classes and their expressions on the one hand, and models of universal algebras and varieties instantiated with equational theories on the other. While occasionally a source of tension in that translation in either direction is not (yet) fully automatic, this duality also opens the door to the possibility

of fully internalized implementations of generic tactics for algebraic manipulation, no longer requiring plugins. One missing piece in this puzzle is automatic quotation of concrete expressions into universal algebra expressions. We have already implemented a proof of concept showing that like unification hints [1], type classes can be used to implement Ltac/OCaml-free quotation.

4 Conclusions

Our development (which according to `coqwc` consists of about 5K lines of specifications and 1K lines of proofs) shows that the first class type class implementation in Coq is already an extremely powerful piece of technology which enables new practical and elegant solutions to old problems.

In our work we push the type class implementation and the new generalized rewriting infrastructure [10] to their limits, revealing both innocent bugs as well as more serious issues (concerning both efficiency and functionality) that the Coq development team is already working on (for instance with the soon to be revealed new proof engine).

Acknowledgements. We would like to thank Matthieu Sozeau for discussions and quickly solving numerous small bugs and feature requests.

References

1. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: Hints in unification. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, p. 98. Springer, Heidelberg (2009)
2. Berardi, S., Damiani, F., de'Liguoro, U. (eds.): TYPES 2008. LNCS, vol. 5497. Springer, Heidelberg (2009)
3. Capretta, V.: Universal algebra in type theory. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 131–148. Springer, Heidelberg (1999)
4. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-coRN, the constructive coq repository at nijmegen. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 88–103. Springer, Heidelberg (2004)
5. Domínguez, C.: Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 270–284. Springer, Heidelberg (2008)
6. 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)
7. Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in Coq. J. Symb. Comput. 34(4), 271–286 (2002)
8. Haftmann, F., Wenzel, M.: Local theory specifications in Isabelle/Isar. In: Berardi, et al. (eds.) [2], pp. 153–168
9. Luo, Z.: Manifest fields and module mechanisms in intensional type theory. In: Berardi, et al. (eds.) [2], pp. 237–255
10. Sozeau, M.: A new look at generalized rewriting in type theory. Journal of Formalized Reasoning 2(1), 41–62 (2009)

Author Index